

# Linux 那些事儿 之我是USB

◎华清远见嵌入式培训中心 肖林甫 肖季东 任桥伟 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

## 内 容 简 介

本书基于 2.6.22 内核，对 USB 子系统的大部分源代码逐行进行分析，系统地阐释了 Linux 内核中 USB 子系统是如何运转的，子系统内部的各个模块之间是如何互相协作互相配合的。

本书使用幽默诙谐的笔调对 Linux 内核中的 USB 子系统源代码进行了分析，形象且详尽地介绍了 USB 在 Linux 中的实现。本书从 U 盘、Hub、USB Core 直到主机控制器覆盖了 USB 实现的方方面面，被一些网友誉为 USB 开发的“圣经”。

对于 Linux 初学者，可以通过本书掌握学习内核、浏览内核代码的方法；对于 Linux 驱动开发者，可以通过本书对设备模型有形象深刻的理解；对于 USB 开发者，可以通过本书全面的理解 USB 在一个操作系统中的实现；对于 Linux 内核开发者，也可以通过本书学习到很多 Linux 高手开发维护一个完整子系统时的编程思想。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

## 图书在版编目（CIP）数据

Linux 那些事儿之我是 USB / 肖林甫, 肖季东, 任桥伟著. -- 北京: 电子工业出版社, 2010.7  
ISBN 978-7-121-11178-5

I. ①L… II. ①肖… ②肖… ③任… III. ①Linux 操作系统—程序设计②电子计算机—接口—程序设计  
IV. ①TP316.89②TP334

中国版本图书馆 CIP 数据核字（2010）第 117276 号

责任编辑：孙学瑛

文字编辑：王 静

印 刷：北京天宇星印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：860×1092 1/16 印张：36 字数：843 千字

印 次：2010 年 7 月第 1 次印刷

印 数：4000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 [zlt@phei.com.cn](mailto:zlt@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：(010) 88258888。

---

# 导 读

## Linux 那些事儿之我是 USB Core

2 - 6, 对 USB 协议规范的简单描述。

7, USB 设备在 sysfs 文件系统中的表示。

8 - 9, 通过对 README、Kconfig、Makefile 文件的分析, 定位要分析的目标代码范围。

10 - 11, USB 子系统的初始化函数 `usb_init()`。

12 - 13, 2.6 内核的设备模型, 以及设备模型在 USB 子系统映射。

14 - 19, USB 子系统实现中的几个重要数据结构。

- 14, `struct usb_interface`, 接口。
- 15, `struct usb_host_interface`, 设置。
- 16, `struct usb_host_endpoint`, 端点。
- 17, `struct usb_device`, 设备。
- 18, `struct usb_host_config`, 配置。
- 19, `struct usb_driver`, USB 接口驱动, `struct usb_device_driver`, USB 设备驱动, 以及 USB 设备与 USB 驱动的匹配函数 `usb_device_match()`。

20 - 30, USB 设备连接到 Hub 上之后, 内核中 USB 子系统的处理过程。

- 20, 设备生命线 (一), `usb_alloc_dev()`, USB 设备的构造函数。
- 21, 设备生命线 (二), `usb_control_msg()`, 创建一个控制 urb, 并把它发送给 USB 设备, 然后等待它完成。
- 22-23, 设备生命线 (三)、设备生命线 (四), `struct urb`, 描述 USB 数据传输的结构。
- 24, 设备生命线 (五), `usb_alloc_urb()`, 创建一个 urb, `usb_fill_control_urb()`, 初始化控制 urb, `usb_start_wait_urb()`, 将 urb 提交给 USB Core, 以便分配给特定的主机控制器驱动进行处理, 然后默默地等待处理结果。
- 25, 设备生命线 (六), `usb_submit_urb`, 启动和停止 USB 数据传输, 对 urb 做些前期处理后扔给 HCD。
- 26-27, 设备生命线 (七)、设备生命线 (八), `struct usb_hcd`, USB 主机控制器驱动, `struct usb_bus`, USB 总线。

- 28, 设备生命线（九），`usb_hcd_submit_urb()`，将提交过来的 `urb` 指派给合适的主机控制器驱动程序。
- 29, 设备生命线（十），`usb_get_device_descriptor()`，获得设备描述符，`usb_get_configuration()`，获得配置描述符。
- 30, 设备生命线（十一），`usb_parse_configuration()`，解析配置描述符。

31-34, USB 驱动从注册到卸载的处理过程。

- 31, 驱动生命线（一），`usb_register_device_driver()`，注册 USB 世界里唯一的那个 USB 设备驱动（不是 USB 接口驱动）`struct device_driver` 结构体对象 `usb_generic_driver`。
- 32-34, 驱动生命线（二）、驱动生命线（三）、驱动生命线（四），`usb_set_configuration()`，配置设备。

35, 字符串描述符。

36, `usb_register()`，注册接口的驱动。

37, `usb_device_match()`，匹配 USB 设备与 USB 驱动。

## Linux 那些事儿之我是 HUB

3, Root Hub, 与 USB 主机控制器集成在一起的根 Hub。`usb_hub_init()`，Hub 的初始化程序。

4, `struct usb_driver hub_driver`，Hub 的接口驱动程序。

5, `hub_thread()`，Hub 驱动中最精华的代码。`struct usb_hub`，Hub 数据结构。`hub_event_list`，Hub 事件链表。

6, `hub_events()`，处理线程 `khubd` 的工作链表 `hub_event_list`。

7, `hub_probe()`，Hub 驱动的 `probe` 函数。

8, 工作队列。

9-13, 配置 Hub 的过程。

- 9, `hub_configure()`，配置 Hub 设备的函数。
- 10, `get_hub_descriptor()`，获得 Hub 描述符。
- 11, `struct usb_tt`，transaction translator 的数据结构，负责 Hub 在高速和低速/全速的数据转换。`hub_hub_status()`，返回 Hub 状态的函数。
- 12-13, `kick_khubd()`，唤醒等待队 `khubd_wait` 上的等待线程 `hub_thread()`。

14, `locktree()`，锁住 USB 设备树。

15 - 16, `hub_port_status()`，返回 Hub 各个端口的状态。`hub_port_connect_change()`，当 Hub 端口上有连接变化时调用这个函数。

18-22, Hub 端口上有连接变化时的处理过程。

- 18, `usb_set_device_state()`, 设置设备状态的函数。
- 19, `choose_address()`, 为设备选择地址的函数。
- 20, `hub_port_init()`, Hub 所含端口的初始化函数。
- 22, `usb_new_device()`, 寻找驱动程序, 调用驱动程序的 `probe`, 跟踪这个函数就能一直跟踪到设备驱动程序的 `probe()` 函数的调用。

23, `hub_power_on()`, Hub 上电。

24, `hub_irq()`, Hub 的中断处理函数, 负责调用 `kick_khubd()`。

## Linux 那些事儿之我是 UHCI

1 - 2, UHCI 驱动的初始化和退出。

- `uhci_hcd_init()`, UHCI 驱动的初始化函数。
- `uhci_hcd_cleanup()`, UHCI 驱动的退出函数。
- `struct pci_driver uhci_pci_driver`, 从 PCI 层面上描述, UHCI 的驱动 (注意 UHCI 是一个 PCI 设备)。

3, `usb_hcd_pci_probe()`, UHCI 驱动的 `probe` 函数。`usb_create_hcd()`, 创建 `struct usb_hcd`。`struct hc_driver uhci_driver`, UHCI 驱动结构。

4, I/O 内存与 I/O 端口。

5, `usb_add_hcd()`, 初始化并注册 `struct usb_hcd`。

6, `usb_register_bus()`, 注册 USB 总线函数。

7, `uhci_init()`, UHCI 初始化。

8, UHCI 的中断资源。

9-13, `uhci_start()`, 初始化 UHCI 的帧列表。`uhci_alloc_td()`, 创建 Transfer Descriptors。`uhci_alloc_qh()`, 创建 Queue Heads。

14, `usb_hcd_poll_rh_status()`, 轮询 Root Hub 的状态变化情况。

15 - 16, Root Hub 的控制传输过程。

- `usb_submit_urb`, 启动和停止 USB 数据传输, 对 `urb` 做些前期处理后扔给 HCD。
- `usb_hcd_submit_urb()`, 主机控制器向设备提交 `urb`。
- `rh_urb_enqueue()`, 对 Root Hub 的 `urb` 进行排队。
- `rh_call_control()`, 向 Root Hub 发送控制命令。

17, `uhci_submit_bulk()`, 非 Root Hub 的批量传输。

- 18, uhci\_irq(), UHCI 的中断服务程序。
- 19, rh\_queue\_status(), Root Hub 的中断传输。
- 20, uhci\_submit\_interrupt(), 非 Root Hub 的中断传输。
- 21, uhci\_submit\_isochronous(), 非 Root Hub 的等时传输。

## Linux 那些事儿之我是 U 盘

- 1, 通过对 Kconfig、Makefile 文件的分析, 定位要分析的目标代码范围。
- 2 - 4, usb\_stor\_init(), USB storage 模块的初始化函数。usb\_stor\_exit(), USB storage 模块的退出函数。
- 5 - 9, 2.6 内核的设备模型, 以及设备模型在 USB 子系统映射。struct usb\_driver usb\_storage\_driver, U 盘驱动结构。
- 10, struct usb\_device\_id, USB 设备的 ID。
- 14, storage\_probe(), USB storage 模块的 probe 函数。struct us\_data, 会为每一个 USB storage 设备申请一个 us\_data。
- 16, associate\_dev(), 为 us\_data 的各个成员赋值。
- 17 - 19, get\_device\_info(), 获得设备信息。
- 20, get\_transport(), 决定用的是哪种传输模式。
- 21, get\_protocol(), 决定用的是哪种通信协议。
- 22, usb\_stor\_acquire\_resources(), 初始化所有需要的动态资源。
- 23-40, struct urb, 传说中的 urb。
- 24- 25, usb\_stor\_control\_thread (), USB storage 模块的内核守护进程, 负责监听命令。
- 26, usb\_stor\_scan\_thread(), Usb storage 模块的内核守护进程, 负责扫描设备。
- 27, queuecommand(), 排队 SCSI 命令。
- 28, struct scsi\_device, SCSI 设备。
- 33, usb\_stor\_show\_command(), 打印 SCSI 命令信息。
- 34 - 40, usb\_stor\_Bulk\_transport(), USB storage 模块中批量传输的函数。
- 45, storage\_disconnect(), USB 设备离开主机时的处理。

---

# 前言

1991 年，Linux 诞生了。又因为开放源代码的缘故，十几年来 Linux 是越来越火，熊熊火焰也烧到了华夏大地。诸多高校开始开设 Linux 相关的课程，诸多企业开始招聘 Linux 相关的人才。市面上关于 Linux 的书籍也层出不穷，而这其中大致分为两类，一类是应用方面的书籍，比如介绍如何组建各种服务器；另一类是内核方面的书籍，主要都是对内核源代码进行分析，这方面的书籍则以《Linux 设备驱动程序》和《深入理解 Linux 内核》为经典代表。而从眼下国内的人才市场来看，懂 Linux 内核的人找工作肯定不用发愁。事实上，毫不夸张地说，当代大学生，如果能够看完以上这两本书，并且能够看懂，那么在北京、上海、深圳这些一线城市，很容易就能找到一份体面的技术类工作。

那么为什么我们还要另起炉灶再写一本 Linux 内核方面的书籍呢？因为 Linux 内核包含大量的代码，以上面两本书为代表的很多 Linux 内核的书籍涵盖的内容太广，大多数书籍都是把 Linux 内核中的各个部分逐一地进行介绍和分析，然而实际情况是没有任何一个人能够对 Linux 内核的各个部分都很精通，包括 Linus Torvalds 本人。一个对 Linux 开发感兴趣的人也不一定需要并且有足够的时间对 Linux 的每个部分都去深入理解。而另一方面，很多对 Linux 内核感兴趣的朋友常常被一个问题所困扰，那就是 Linux 内核那么庞大的代码量，对于初学者来说，应该从哪里学起呢？关于这一点，其实《Linux 设备驱动程序》给出了很好的答案，学习驱动程序代码是最有效的入门方法。第一是因为在庞大的 Linux 内核源代码中，大约 87.53% 是各种驱动程序代码，其重要性可想而知，第二是因为相对来说，驱动程序的代码其难度是比较低的，很少涉及复杂高深的算法，所以适合初学者研读。

然而网友“永不堕落”曾经问过我们：“既然已经有了《Linux 设备驱动程序》，为什么你们还要写一本 Linux 设备驱动程序相关的书呢？你们这样做是不是行为艺术呢？”这里我们想说的是，虽然《Linux 设备驱动程序》这本书很强大了，把各种类型的设备驱动程序都给介绍了一番，可是当一些读者读完这本书之后，他们敢说会写 Linux 设备驱动程序了吗？他们敢说完全了解一个真实的 Linux 设备驱动程序是怎么写的吗？至少本书作者当年在看完这本书之后，虽然觉得获益匪浅，可是仍然不太清楚真实的 Linux 设备驱动程序是怎么写的，仍然不敢认为自己就会写 Linux 设备驱动程序了。这才有了后来决定亲自选择一个子系统进行研

究，并在研究好了之后把研究心得写出来，与大家进行分享。最终我们选择的是 USB 子系统，原因很简单，USB 总线及连接在 USB 总线上的各种 USB 设备已经广泛地出现在了当代计算机上，广大计算机用户，尤其是高校学生，接触得最多的设备也正是 USB 设备，所以研究和分析 USB 总线，以及它上面的各种 USB 设备应该能让大家感到很亲切很真实并且具有相当的实用价值和怀旧意义。有人曾经说过：“USB 总线就像一条河，左岸是我无法忘却的回忆，右岸是我值得紧握的璀璨年华，中间流淌的，是我年年岁岁淡淡的感伤！”

本书的编写得到了许多人的帮助，在此向他们致以诚挚的谢意。首先感谢孙学瑛编辑，没有她的努力，这本书的内容将会一直偏居网络一隅，将不可能被出版从而去帮助更多需要的人。然后要感谢很多在技术上给予我们指导与帮助的老师和朋友，特别是 USB 这边的 maintaner 之一，Alan Stem 大侠对我们的问题的耐心解答与回复。



# 目 录

## 第 1 篇 Linux 那些事儿之我是 USB Core

1. 引子.....	2	20. 设备的生命线（一）.....	53
2. 它从哪里来.....	2	21. 设备的生命线（二）.....	56
3. PK.....	3	22. 设备的生命线（三）.....	61
4. 漫漫辛酸路.....	3	23. 设备的生命线（四）.....	67
5. 我型我秀.....	4	24. 设备的生命线（五）.....	73
6. 我是一棵树.....	5	25. 设备的生命线（六）.....	80
7. 我是谁.....	9	26. 设备的生命线（七）.....	88
8. 好戏开始了.....	11	27. 设备的生命线（八）.....	94
9. 不一样的 Core.....	13	28. 设备的生命线（九）.....	100
10. 从这里开始.....	17	29. 设备的生命线（十）.....	104
11. 面纱.....	20	30. 设备的生命线（十一）.....	109
12. 模型，又见模型.....	22	31. 驱动的生命线（一）.....	122
13. 繁华落尽.....	26	32. 驱动的生命线（二）.....	127
14. 接口是设备的接口.....	28	33. 驱动的生命线（三）.....	131
15. 设置是接口的设置.....	32	34. 驱动的生命线（四）.....	135
16. 端点.....	35	35. 字符串描述符.....	138
17. 设备.....	37	36. 接口的驱动.....	147
18. 配置.....	45	37. 还是那个 match.....	150
19. 向左走，向右走.....	48	38. 结束语.....	155

## 第 2 篇 Linux 那些事儿之我是 HUB

1. 引子.....	157	4. 一样的精灵不一样的 API.....	160
2. 跟我走吧，现在就出发.....	157	5. 那些队列，那些队列操作函数.....	164
3. 特别的爱给特别的 Root Hub.....	158	6. 等待，只因曾经承诺.....	169

7. 最熟悉的陌生人——probe	171	16. 一个都不能少	206
8. 蝴蝶效应	174	17. 盖茨家对 Linux 代码的影响	215
9. While You Were Sleeping (一)	178	18. 八大重量级函数闪亮登场 (一)	220
10. While You Were Sleeping (二)	183	19. 八大重量级函数闪亮登场 (二)	223
11. While You Were Sleeping (三)	185	20. 八大重量级函数闪亮登场 (三)	225
12. While You Were Sleeping (四)	191	21. 八大重量级函数闪亮登场 (四)	237
13. 再向虎山行	194	22. 八大重量级函数闪亮登场 (五)	241
14. 树, 是什么样的树	198	23. 是月亮惹的祸还是 spec 的错	249
15. 没完没了的判断	201	24. 所谓的热插拔	251

### 第 3 篇 Linux 那些事儿之我是 UHCI

1. 引子	256	12. 一个函数引发的故事 (四)	309
2. 开户和销户	258	13. 一个函数引发的故事 (五)	311
3. PCI, 我们来了!	262	14. 寂寞在唱歌	313
4. I/O 内存和 I/O 端口	270	15. Root Hub 的控制传输 (一)	321
5. 传说中的 DMA	275	16. Root Hub 的控制传输 (二)	327
6. 来来, 我是一条总线, 线线线线线线	281	17. 非 Root Hub 的批量传输	339
7. 主机控制器的初始化	285	18. 传说中的中断服务程序 (ISR)	345
8. 有一种资源, 叫中断	293	19. Root Hub 的中断传输	362
9. 一个函数引发的故事 (一)	295	20. 非 Root Hub 的中断传输	364
10. 一个函数引发的故事 (二)	298	21. 等时传输	375
11. 一个函数引发的故事 (三)	303	22. “脱”就一个字	381

### 第 4 篇 Linux 那些事儿之我是 U 盘

1. 小城故事	388	11. 从协议中来, 到协议中去 (上)	401
2. Makefile	389	12. 从协议中来, 到协议中去 (中)	403
3. 变态的模块机制	390	13. 从协议中来, 到协议中去 (下)	405
4. 想到达明天现在就要启程	392	14. 梦开始的地方	406
5. 外面的世界很精彩	394	15. 设备花名册	411
6. 未曾开始却似结束	395	16. 冰冻三尺非一日之寒	412
7. 狂欢是一群人的孤单	396	17. 冬天来了, 春天还会远吗? (一)	416
8. 总线、设备和驱动 (上)	397	18. 冬天来了, 春天还会远吗? (二)	422
9. 总线、设备和驱动 (下)	398	19. 冬天来了, 春天还会远吗? (三)	425
10. 我是谁的他	400	20. 冬天来了, 春天还会远吗? (四)	427

21. 冬天来了，春天还会远吗？（五）	431	35. 迷雾重重的批量传输（二）	476
22. 通往春天的管道	436	36. 迷雾重重的批量传输（三）	479
23. 传说中的 URB	440	37. 迷雾重重的批量传输（四）	484
24. 彼岸花的传说（一）	443	38. 迷雾重重的批量传输（五）	489
25. 彼岸花的传说（二）	445	39. 迷雾重重的批量传输（六）	493
26. 彼岸花的传说（三）	448	40. 迷雾重重的批量传输（七）	495
27. 彼岸花的传说（四）	451	41. 跟着感觉走（一）	500
28. 彼岸花的传说（五）	453	42. 跟着感觉走（二）	503
29. 彼岸花的传说（六）	457	43. 有多少爱可以胡来？（一）	509
30. 彼岸花的传说（七）	460	44. 有多少爱可以胡来？（二）	513
31. 彼岸花的传说（八）	463	45. 当梦醒了天晴了	518
32. 彼岸花的传说（The End）	467	46. 其实世上本有路，走的人多了， 也便没了路	522
33. SCSI 命令之我型我秀	468		
34. 迷雾重重的批量传输（一）	472		

附录   Linux 那些事儿之我是 sysfs

1. sysfs 初探	526	3.1.2 一起散散步 path_walk	551
2. 设备模型	527	3.1.3 super_block 与 vfstmount	552
2.1 设备底层模型	528	3.2 sysfs	553
2.1.1 kobject	528	3.2.1 sysfs_dirent	553
2.1.2 kset	530	3.2.2 sysfs_create_dir()	554
2.1.3 kobj_type	531	3.2.3 sysfs_create_file()	556
2.2 设备模型上层容器	532	3.3 file_operations	557
2.3 示例一：usb 子系统	535	3.3.1 示例一：读入 sysfs 目录的 内容	558
2.4 示例二：usb storage 驱动	540	3.3.2 示例二：读入 sysfs 普通 文件的内容	560
3. sysfs 文件系统	546		
3.1 文件系统	547		
3.1.1 dentry & inode	548		

# 第 1 篇

## Linux 那些事儿之我是 USB Core

1. 引子.....	2	20. 设备的生命线（一）.....	53
2. 它从哪里来.....	2	21. 设备的生命线（二）.....	56
3. PK .....	3	22. 设备的生命线（三）.....	61
4. 漫漫辛酸路.....	3	23. 设备的生命线（四）.....	67
5. 我型我秀.....	4	24. 设备的生命线（五）.....	73
6. 我是一棵树.....	5	25. 设备的生命线（六）.....	80
7. 我是谁.....	9	26. 设备的生命线（七）.....	88
8. 好戏开始了.....	11	27. 设备的生命线（八）.....	94
9. 不一样的 Core.....	13	28. 设备的生命线（九）.....	100
10. 从这里开始.....	17	29. 设备的生命线（十）.....	104
11. 面纱.....	20	30. 设备的生命线（十一）.....	109
12. 模型，又见模型.....	22	31. 驱动的生命线（一）.....	122
13. 繁华落尽.....	26	32. 驱动的生命线（二）.....	127
14. 接口是设备的接口 .....	28	33. 驱动的生命线（三）.....	131
15. 设置是接口的设置 .....	32	34. 驱动的生命线（四）.....	135
16. 端点.....	35	35. 字符串描述符.....	138
17. 设备.....	37	36. 接口的驱动.....	147
18. 配置.....	45	37. 还是那个 match.....	150
19. 向左走，向右走.....	48	38. 结束语.....	155

---

## 1. 引子

老夫子们痛心疾首地总结说，现代青年的写照——自负太高，反对太多，商议太久，行动太迟，后悔太早。上天戏弄，我不幸地混进了“80 后”的革命队伍里，成了一名现代青年，前有老夫子的忧心忡忡，后有“90 后”的轻蔑嘲弄，终日在“迷失”与“老土”这样的两极词汇里徘徊。

这里我就讲一讲 USB，让他们看一看“80 后”还知道什么叫 USB。

还是要说在前面，在这里耗费青春写 USB，并不是因为喜欢它，相反，对它毫无感觉可言，虽然每天都必须和它相依为伴，不离不弃，不过那可是丝毫没有办法的事情，非我所愿。是不是特说到心坎儿里去了？不过您别多想，咱这里只谈 USB。

一句话总结：哥写的不是 USB，是寂寞。

---

## 2. 它从哪里来

“你从哪里来，我的朋友，好像一只蝴蝶，飞进我的窗口。”

在嘹亮的歌声中，USB 好像一只蝴蝶飞进了千家万户。它从哪里来，它从 Intel 来。Intel 不养蝴蝶，而是做 CPU，它只是在蝴蝶的翅膀上烙上“Intel inside”，蝴蝶让咱们的同胞去养了，然后带着 Intel 飞进了千万家。

不过，与 PCI、AGP 属于 Intel 单独提出的硬件标准不同，Compaq、IBM、Microsoft 等也一起参与了 USB 这个游戏。他们一起于 1994 年 11 月提出了 USB，并于 1995 年 11 月制定了 0.9 版本，1996 年制定了 1.0 版本。不过 USB 并没有因为有这些大佬的支持立即迎来它的春天，只怪它诞生在了冬季，生不逢时啊！

因为缺乏操作平台的良好支持和大量支持它的产品，这些标准都成了空谈。1998 年 USB1.1 的出现，忽如一夜春风来，它就像春天里的一朵油菜花，终于涂上了浓重的一抹黄色。

为什么要开发 USB？

在 USB 出现以前，电脑的接口处于“春秋战国时代”，串口并口等多方割据，键盘、鼠标、MODEM、打印机、扫描仪等都要连接在这些不同种类的接口上，一个接口只能连接一个设备。

不过咱们的电脑不可能有那么多接口，所以扩展能力不足，而且速度也确实很有限。还有关键的一点是，热插拔对它们来说也是比较危险的操作。

USB 正是为了解决速度、扩展能力、易用性等问题应景而生的。

---

### 3. PK

在 2006 年，最火的是“超级女生”，最流行的词是“PK”。

USB 的一生也充满了“PK”，不过 USB 还不够老，说一生还太早了，发哥说得好：“我才刚上路呢！”

USB 最初的设计目标就是替代串行、并行等各种低速总线，以一种单一类型的总线连接各种不同的设备。它现在几乎可以支持所有连接到 PC 上的设备，1999 年提出的 USB 2.0 理论上可以达到 480 MB/s 的速度，2008 年公布的 USB 3.0 标准更是提供了十倍于 USB 2.0 的传输速度。

因此，USB 与串口、并口等的这场“PK”从一开始就是不平等的，这样的开始也注定了以什么样的结果结束，只能说命运选择了 USB。我们很多人都说命运掌握在自己手里，但是从 USB 充满“PK”的一生中可以知道，只有变得比别人更强，命运才能掌握在自己手里。

有了 USB 在这场 PK 中的大获全胜，才有了 USB 键盘、USB 鼠标、USB 打印机、USB 摄像头、USB 扫描仪、USB 音箱等。至于将来，“PK 自己的，让别人去说吧！”USB 如是说。

---

### 4. 漫漫辛酸路

USB 的一生充满了“PK”，并在“PK”中发展，从 USB 1.0、USB 1.1、USB 2.0 到 USB 3.0，漫漫辛酸路，一把辛酸泪。

USB 2.0 的高速模式（High-Speed）最高已经达到了 480 MB/s，也就是说，以这个速度，你将自己从网上下载的短片备份到自己的移动硬盘上的时间长约为一秒钟。而 USB 3.0 的 Super-Speed 模式比这个速度还要提高了几乎 10 倍，达到了 4.8GB/s。

USB 走过的这段辛酸路，对咱们来说最直观的结果也就是传输速度提高了，过程很艰辛，结果很简单。

USB 的各个版本都是兼容的。每个 USB 2.0 控制器带有 3 个芯片，根据设备的识别方式将信号发送到正确的控制芯片。我们可以将 USB 1.1 设备连接到 USB 2.0 的控制器上使用，不过它只能达到 USB 1.1 的速度。同时也可以将 USB 2.0 的设备连接到 USB 1.1 的控制器上，不过不能指望它能以 USB 2.0 的速度运行。

显然，Linux 对 USB1.1 和 USB 2.0 都是支持的，并抢在 Windows 前，在 2.6.31 内核中率先对 USB 3.0 进行了支持。

---

## 5. 我型我秀

USB 既然能一路“PK”走过来，也算是一个挺能“秀”的角色了，不然也不会有那么多的拥护者。

USB 为所有 USB 外设都提供了单一的标准连接类型，这就简化了外设的设计，也让我们不用再去想哪个设备对应哪个插槽的问题，就像种萝卜，一个萝卜一个坑，但是哪个萝卜种到哪个坑里是不用我们关心的。

USB 支持热插拔，而其他的比如 SCSI 设备等只有在关掉主机的前提下才能增加或移走外围设备。所以说，USB 的一生不仅仅是“PK”的一生，也是丰富多彩的一生，可以不用关机就能更换不同种类的外设。

USB 在设备供电方面提供了灵活性。USB 设备可以通过 USB 电缆供电，不然移动硬盘、iPod 等常备外设也用不了了。相对应，有的 USB 设备也可以使用普通的电源供电。

USB 能够支持从每秒几十 KB 到几十 MB 的传输速率，来适应不同种类的外设。它可以支持多个设备同时操作，也支持多功能的设备。多功能的设备当然指的就是一个设备同时有多个功能，比如 USB 扬声器。这通过在一个设备中包含多个接口来支持，一个接口支持一个功能。

USB 可以支持多达 127 个设备。

USB 可以保证固定的带宽，这个对视频音频设备是利好。

## 6. 我是一棵树

“我是一棵树，静静地站在田野里，风儿吹过，我不知它的去向，人儿走过，我不知谁会为我停留。”

如图 1.6.1 所示，USB 子系统的拓扑也是一棵树，它并不以总线的方式来部署。

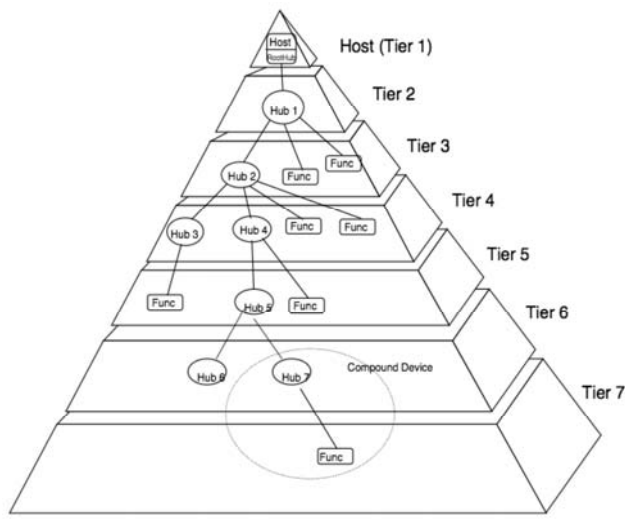


图 1.6.1 USB 子系统的树形结构

我曾经指着路边一棵奇形怪状的树问朋友：“这是什么树？”朋友的回答让我很晕：“大树。”那图 1.6.1 指的是什么树？自然也是大树了，不过却是 USB 的大树。这棵大树主要包括了 USB 连接、USB Host Controller（USB 主机控制器）和 USB 设备三个部分。而 USB 设备还包括了 Hub 和功能设备（也就是上图里的 Func）。

什么是 USB 主机控制器？控制器，顾名思义，用于控制。控制什么？控制所有的 USB 设备的通信。通常计算机的 CPU 并不是直接和 USB 设备打交道，而是和控制器打交道。它要对设备做什么，它会告诉控制器，而不是直接把指令发给设备。然后控制器再去负责处理这件事情，它会去指挥设备执行命令，而 CPU 就不用管剩下的事情。控制器替他去完成剩下的事情，事情办完了再通知 CPU。否则让 CPU 去盯着每一个设备做每一件事情，那是不现实的。

这就好像让一个学院的院长去盯着我们每一个学生上课，管理学生的出勤，这是不现实的。所以学生就被分成了几个系，通常院长有什么指示直接跟各系领导说就可以了，如果他要和三个系主任说事情，他即使不把三个人都召集起来开会，也可以给三个人各打一个电话，打完电话他就忙他自己的事情去了。而三个系主任就会去安排下面的人去执行具体的任务，然后他们就会向院长汇报。



那么 Hub 是什么？在大学里，有的宿舍里网口有限，所以会有网口不够用的情况出现，于是有人会使用 Hub，让多个人共用一个网口，这是以太网上的 Hub。而 USB 的世界里同样有 Hub，其实原理是一样的，任何支持 USB 的计算机不会只允许你只能一个时刻使用一个 USB 设备，比如你插入了 U 盘，同样还可以插入 USB 键盘，还可以再插一个 USB 鼠标，因为你会发现你的计算机里并不只是一个 USB 接口。这些接口实际上就是所谓的 Hub 口。

而现实中经常是一个 USB 控制器和一个 Hub 绑定在一起，专业一点称为“集成”，而这个 Hub 也被称做 Root Hub。换言之，和 USB 控制器绑定在一起的 Hub 就是系统中最根本的 Hub，其他的 Hub 可以连接到它这里，然后可以延伸出去，外接别的设备，当然也可以不用别的 Hub，让 USB 设备直接接到 Root Hub 上。

而 USB 连接指的就是连接 USB 设备和主机（或 Hub）的四线电缆。电缆中包括 VBUS（电源线）、GND（地线）还有两根信号线。USB 系统就是通过 VBUS 和 GND 向 USB 设备提供电源的。主机对连接的 USB 设备提供电源供其使用，而每个 USB 设备也能够有自己的电源，如图 1.6.2 所示。

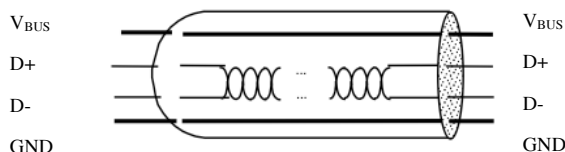


图 1.6.2 USB 四线电缆

现在，如图 1.6.3 所示的 USB 大树里只有 Compound Device 还没有说。那么，Compound Device 又是什么样的设备？其实，在 USB 的世界里，不仅仅有 Compound Device，还有 Composite Device，简单的中文名字已经无法形象地表达它们的区别。正如图 1.6.3 所示，Compound Device 是将 Hub 和连在 Hub 上的设备封装在一起所组成的设备。而 Composite Device 则是包含彼此独立的多个接口的设备。从主机的角度看，一个 Compound Device 和单独的一个 Hub 然后连接了多个 USB 设备是一样的，它里面包含的 Hub 和各个设备都会有自己独立的地址，而一个 Composite Device 里不管有多少接口，它都只有一个地址。

USB 大树要想茁壮成长离不开 USB 协议。USB 总线是一种轮询式总线。协议规定所有的数据传输都必须由主机发起，由主机控制器初始化所有数据传输，各种设备紧紧围绕在主机周围。

USB 通信最基本的形式是通过 USB 设备中一个叫 Endpoint（端点）的东西，而主机和端点之间的数据传输是通过 Pipe（管道）。

端点就是通信的发送点或者接收点，要发送数据，只需把数据发送到正确的端点那里就可以了。而管道，实际上只是为了让我们能够找到端点，就相当于我们日常说的邮编地址。

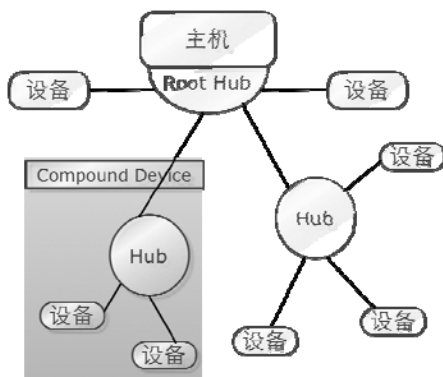


图 1.6.3 Compound Device

比如一个国家，为了通信，我们必须给各个地方取名，然后给各条大大小小的路取名。严格来说，管道的另一端应该是 USB 主机，USB 协议也是这么说的，协议说管道代表着在主机和设备上的端点之间移动数据的能力。

端点不但是有方向的，而且这个方向还是确定的，或者是 in，或者是 out，没有又是 in 又是 out 的，都是生来就注定的。

有没有特殊的端点呢？看你怎么去理解 0 号端点了，协议规定了，所有的 USB 设备必须具有端点 0，它可以作为 in 端点，也可以作为 out 端点。USB 系统软件利用它来实现默认的控制管道，从而控制设备。

端点也是限量供应的，不是想有多少就有多少，除了端点 0，低速设备最多只能拥有两个端点，高速设备也最多只能拥有 15 个 in 端点和 15 个 out 端点。这些端点在设备内部都有唯一的端点号，这个端点号是在设备设计时就已经指定的。

为什么端点 0 就特殊呢？这还是有内在原因的。管道的通信方式其实有两种：一种是 stream 的，一种是 message 的。message 管道要求从它那儿过的数据必须具有一定的格式，不是随便传的，因为它主要就是用于主机向设备请求信息的，必须得让设备明白请求的是什么。而 stream 管道就没这么苛刻，随和多了，对数据没有特殊的要求。协议中规定，message 管道必须对应两个相同号码的端点：一个用来 in，一个用来 out，默认管道就是 message 管道。当然，与默认管道对应的端点 0 就必须是两个具有同样端点号 0 的端点。

USB 端点有四种类型，也就分别对应了四种不同的数据传输方式。它们是控制传输 (Control Transfers)、中断传输 (Interrupt Data Transfers)、批量传输 (Bulk Data Transfers)，等时传输 (Isochronous Data Transfers)。控制传输用来控制对 USB 设备不同部分的访问，通常用于配置设备，获取设备信息，发送命令到设备，或者获取设备的状态报告。总之就是用来传送控制信息的，每个 USB 设备都会有一个名为“端点 0”的控制端点，内核中的 USB Core 使用它在设

备插入时进行设备的配置。

中断传输用来以一个固定的速率传送少量的数据，USB 键盘和 USB 鼠标使用的就是这种方式，USB 的触摸屏也是使用这种方式，传输的数据包含了坐标信息。

批量传输用来传输大量的数据，确保没有数据丢失，但不保证在特定的时间内完成。U 盘使用的就是批量传输，用它备份数据时需要确保数据不能丢，而且也不能指望它能在一个固定的比较快的时间内复制完。

等时传输同样用来传输大量的数据，但并不保证数据是否到达，以稳定的速率发送和接收实时的信息，对传送延迟非常敏感，显然是用于音频和视频一类的设备。这类设备期望能够有一个比较稳定的数据流，比如在用 QQ 视频聊天时，肯定希望每分钟传输的图像/声音速率是比较稳定的，不能说这一分钟前对方看到你在向她向你深情表白，可是下一分钟却看见画面停滞在那里，只能看到你在那里一动不动，这不是浪费感情吗！

如图 1.6.1 所示的树形结构描述的是实实在在的物理拓扑，对于内核中的实现来说，没有这么复杂，所有的 Hub 和设备都被看做是一个个的逻辑设备 (Logical Device)，如图 1.6.4 所示，好像它们本来就直接连接在 Root Hub 上一样。

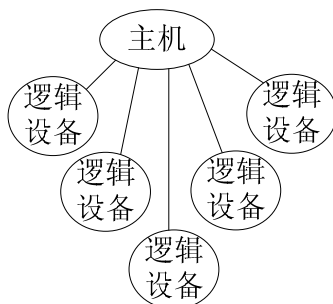


图 1.6.4 USB 逻辑拓扑结构

如图 1.6.5 所示，一个 USB 逻辑设备就是一系列端点的集合，它与主机之间的通信发生在主机上的一个缓冲区和设备上的一个端点之间，通过管道来传输数据。也就是说管道的一端是主机上的一个缓冲区，一端是设备上的端点。

那么图 1.6.5 中的接口又是指什么？简单地说，USB 端点被捆绑为接口 (Interface)，一个接口代表一个基本功能。有的设备具有多个接口，像 USB 扬声器就包括一个键盘接口和一个音频流接口。在内核中一个接口要对应一个驱动程序，USB 扬声器在 Linux 里就需要两个不同的驱动程序。到目前为止，一个设备可以包括多个接口，一个接口可以具有多个端点，当然以后我们会发现并不仅仅止于此。

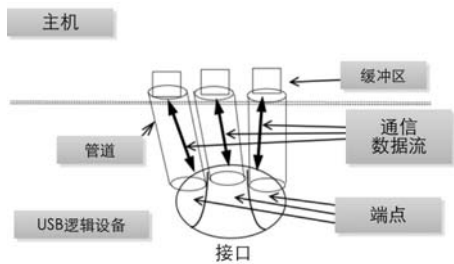


图 1.6.5 USB 数据通信

## 7. 我是谁

我是谁？USB 也一遍一遍地问着自己，当然它不会真的是一棵树，它也不会是太阳，Linux 里没有太阳，真要有的话也只能是 Linus。USB 子系统只是 Linux 庞大家族里的一个小部落，主机控制器是它们的族长，族里的每个 USB 设备都需要被系统识别，被我们识别，而 sysfs 就是它们对外的窗口，我们可以从 sysfs 里了解认识每一个 USB 设备。以一个仅包含一个 USB 接口的 USB 鼠标为例，如图 1.7.1 所示就是该设备对应的 sysfs 目录树。

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   |-- power
|   |-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|-- state
|-- speed
|-- version
```

图 1.7.1 USB 鼠标的 sysfs 目录树

其中：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
```

表示鼠标。

下层目录：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0
```

表示鼠标的 USB 接口。sysfs 里 USB 设备都是类似的表示，设备的目录下包括了表示设备接口的目录。目录里的各个文件表示了设备或接口的描述，大都对应了设备描述符、接口描述符等相应值，可以通过这些值获得您感兴趣的信息。什么是设备描述符还有接口描述符？我们这里要暂时忽略它的存在，先关心关心 USB 设备在 sysfs 里是如何命名的，弄清它是谁，也就是说弄清上面路径的含义。

USB 系统中的第一个 USB 设备是 Root Hub，前面已经说了它是和主机控制器绑定在一起的。这个 Root Hub 通常包含在 PCI 设备中，是连接 PCI 总线和 USB 总线的 bridge，控制着连接到其上的整个 USB 总线。所有的 Root Hub，内核的 USB Core 都分配有独特的编号，在上面的例子里就是 USB 2。

USB 总线上的每个设备都以 Root Hub 的编号作为其名字的第一个号码。这个号码后跟着一个“-”字符，以及设备所插入的端口号。因此，上面例子中的 USB 鼠标的设备名就是 2-1。因为该 USB 鼠标具有一个接口，导致了另外一个 USB 设备被添加到 sysfs 路径中。因为物理 USB 设备和单独的 USB 接口在 sysfs 中都将表示为单独的设备。USB 接口的命名是设备名直到该接口，上面就是 2-1 后面跟一个“:”和 USB 配置（Configuration）的编号，然后是一个“.”和该接口的编号。因此上面的鼠标 USB 接口就是 2-1:1.0，表示使用的是第一个配置，接口编号为 0。

sysfs 并没有展示 USB 设备的所有部分，设备可能包含的可选配置都没有显示，不过这些可以通过 usbfs 找到，该文件系统被挂在 /proc/bus/usb 目录中，从 /proc/bus/usb/device 文件可以知道系统中存在的所有 USB 设备的可选配置。

这里既然提到了 USB 设备的配置，就还是先简要说一下。一个设备可以有一种或者几种配置，这能理解吧？没见过具体的 USB 设备？那么手机见过吧，每部手机都会有多种配置，或者说“设定”。比如，我的这款 Nokia 6300 手机，手机语言可以设定为 English、繁体中文、简体中文，一旦选择了其中一种，那么手机里边所显示的所有信息都是该种语言/字体。再举一个最简单的例子，手机的操作模式也有好几种，标准、无声、会议等。如果我设为“会议”模式，那么就是只振动不发声：要是设为“无声”模式，那么就什么动静也不会有。那么 USB 设备的配置也是如此，不同的 USB 设备当然有不同的配置了，或者说需要配置哪些东西也会不一样。

## 8. 好戏开始了

首先要去 `drivers/usb` 目录下走一走看一看。

```
atm/ class/ core/ gadget/ host/ image/ misc/ mon/ serial/ storage/
Kconfig Makefile README usb-skeleton.c
```

`ls` 命令的结果就是上面的 10 个目录和 4 个文件。`usb-skeleton.c` 是一个简单的 USB driver 的框架。那么首先应该关注什么？那就是 `Kconfig`、`Makefile`、`README`。

`README` 里有关于这个目录下内容的一般性描述。再说了，面对“读我吧！读我吧！”这么热情奔放的呼唤，善良的我们是不可能无动于衷的，所以先来看一看 `README` 里面都有些什么内容。

```
Here is a list of what each subdirectory here is, and what is contained in
them.
```

```
core/          - This is for the core USB host code, including the
                usbfs files and the hub class driver ("khubd").
```

```
host/          - This is for USB host controller drivers. This
                includes UHCI, OHCI, EHCI, and others that might
                be used with more specialized "embedded" systems.
```

```
gadget/        - This is for USB peripheral controller drivers and
                the various gadget drivers which talk to them.
```

```
Individual USB driver directories. A new driver should be added to the
first subdirectory in the list below that it fits into.
```

```
image/         - This is for still image drivers, like scanners or
                digital cameras.
```

```
input/         - This is for any driver that uses the input subsystem,
                like keyboard, mice, touchscreens, tablets, etc.
```

```
media/         - This is for multimedia drivers, like video cameras,
                radios, and any other drivers that talk to the v4l
                subsystem.
```

```
net/           - This is for network drivers.
```

```
serial/        - This is for USB to serial drivers.
```

```
storage/       - This is for USB mass-storage drivers.
```

```
class/         - This is for all USB device drivers that do not fit
                into any of the above categories, and work for a range
                of USB Class specified devices.
```

```
misc/          - This is for all USB device drivers that do not fit
                into any of the above categories.
```

`drivers/usb/README` 文件描述了前边使用 `ls` 命令列出的那 10 个文件夹的用途。那么什么是 USB Core？Linux 内核开发人员们专门写了一些代码，负责实现一些核心的功能，为别的设备驱动程序提供服务，比如申请内存，实现一些所有的设备都会需要的公共的函数，并美其名曰为“USB Core”。

时代总在发展，早期的 Linux 内核，其结构并不是如今天这般的层次感，远不像今天这般错落有致，那时候 `drivers/usb/` 这个目录下放了很多文件，USB Core 与其他各种设备驱动程序代码都堆砌在这里，后来，在 `drivers/usb/` 目录下面出来了一个 `core` 目录，就专门放一些核心的代码，比如初始化整个 USB 系统，初始化 Root Hub，初始化主机控制器的代码，再后来甚至把主机控制器相关的代码也单独建了一个目录，叫 `host` 目录。这是因为 USB 主机控制器随着时代的发展，也开始有了好几种，不再像刚开始那样只有一种。所以呢，设计者们把一些主机控制器公共的代码仍然留在 `core` 目录下，而一些各主机控制器单独的代码则移到 `host` 目录下面让负责各种主机控制器的人去维护。

那么 USB gadget 呢？gadget 说白了就是配件的意思，主要就是一些内部运行 Linux 的嵌入式设备，比如 PDA，设备本身有 USB 设备控制器（USB Device Controller），可以将 PC，也就是我们的主机作为 master 端，将这样的设备作为 slave 端和主机通过 USB 进行通信。从主机的观点来看，主机系统的 USB 驱动程序控制插入其中的 USB 设备，而 USB gadget 的驱动程序控制外围设备作为一个 USB 设备和主机通信。比如，我们的嵌入式主板上支持 SD 卡，如果我们在将主板通过 USB 连接到 PC 之后，这个 SD 卡被模拟成 U 盘，那么就要通过 USB gadget 架构的驱动。

`gadget` 目录下大概能够分为两个模块：一个是 `udc` 驱动，这个驱动是针对具体 cpu 平台的，如果找不到现成的，就要自己实现；另外一个就是 `gadget` 驱动，主要有 `file_storage`、`ether`、`serial` 等。另外还提供了 USB gadget API，即 USB 设备控制器硬件和 `gadget` 驱动通信的接口。PC 及服务器只有 USB 主机控制器硬件，它们并不能作为 USB gadget 存在，而对于嵌入式设备，USB 设备控制器常被集成到处理器中，设备的各种功能，如 U 盘、网卡等常依赖这种 USB 设备控制器来与主机连接，并且设备的各种功能之间可以切换，比如可以选择作为 U 盘或网卡等。

剩下的几个目录分门别类地放了各种 USB 设备的驱动，U 盘的驱动在 `storage` 目录下，触摸屏和 USB 键盘鼠标的驱动在 `input` 目录下等。另外，在 USB 协议中，除了通用的软硬件电气接口规范等，还包含了各种各样的 Class 协议，用来为不同的功能定义各自的标准接口和具体的总线上的数据交互格式和内容。这些 Class 协议的数量非常多，比如最常见的支持 U 盘功能的 `Mass Storage Class`，以及通用的数据交换协议 `CDC Class`。此外还包括 `Audio Class`、`Print Class` 等。理论上讲，即使没有这些 Class，通过专用驱动也能够实现各种各样的应用功能。但是，正是 `Mass Storage Class` 的使用，使得各个厂商生产的 U 盘都能通过操作系统自带的统一驱动程序来使用，对 U 盘的普及使用起了极大的推动作用，制定其他这些 Class 也是同样的目的。

我们响应了 README 的呼唤，它给予了我们想要的，通过它我们了解了 USB 目录里的那些文件夹都有着什么样的角色。到现在为止，就只剩下 `Kconfig` 和 `Makefile` 两个文件了，它们又扮演着什么样的角色？就好像我吃东西时总是喜欢把好吃的留在最后享受一样，我也习惯于将重要的内容留在最后去描述。对于一个希望能够在 Linux 内核的汪洋代码里看到一丝曙光的人来说，将它们放在多么重要的地位都不过分。我们在去香港通过海关时，总会有免费的地图，

有了它们我们才不至于像无头苍蝇般迷惘地行走在陌生的街道上。即使在出去旅游时一般也总是会首先找一份地图，当然了，这时就是要去买，拿是拿不到的。不同的地方有不同的特色，别人的特色是服务，咱们的特色是索取。Kconfig、Makefile 就是 Linux kernel 迷宫里的地图，我们每次浏览 kernel 寻找属于自己的那一段代码时，都应该首先看一看目录下的这两个文件。

不过，这里很明显，要想了解 USB 协议在内核中的实现，USB Core 就是我们需要关注的对象。

## 9. 不一样的 Core

我们来看 core 目录。关于 USB，有一个很重要的模块，它的名字耐人寻味，usbcore。如果你的电脑是装了 Linux 操作系统，那么你用 lsmod 命令看一下，有一个模块叫做 usbcore。当然，你要是玩嵌入式系统的高手，那么也许你的电脑里没有 USB 模块。不过我听说如今玩嵌入式的人也喜欢玩 USB，因为 USB 设备很符合嵌入式的口味。查看一下我的 lsmod 命令的输出吧。

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # lsmod
Module                Size  Used by
af_packet             55820  2
raw                   89504  0
nfs                   230840  2
lockd                 87536  2 nfs
nfs_acl               20352  1 nfs
sunrpc               172360  4 nfs,lockd,nfs_acl
ipv6                  329728  36
button                24224  0
battery               27272  0
ac                    22152  0
apparmor              73760  0
aamatch_pcre         30720  1 apparmor
loop                  32784  0
usbhid                60832  0
dm_mod                77232  0
ide_cd                57120  0
hw_random             22440  0
ehci_hcd              47624  0
cdrom                 52392  1 ide_cd
uhci_hcd              48544  0
shpchp                61984  0
bnx2                  157296  0
usbcore              149288  4 usbhid,ehci_hcd,uhci_hcd
e1000                 130872  0
pci_hotplug           44800  1 shpchp
reiserfs              239616  2
edd                   26760  0
fan                   21896  0
thermal               32272  0
```



```
processor          50280  1 thermal
qla2xxx            149484 0
firmware_class    27904  1 qla2xxx
scsi_transport_fc  51460  1 qla2xxx
sg                52136  0
megaraid_sas      47928  3
piix               27652  0 [permanent]
sd_mod            34176  4
scsi_mod          163760  5 qla2xxx,scsi_transport_fc,sg,megaraid_sas,sd_mod
ide_disk          32768  0
ide_core          164996  3 ide_cd,piix,ide_disk
```

找到了 `usbcore` 那一行吗？它就是这里要说的 USB 子系统的核心，如果要在 Linux 里使用 USB，这个模块是必不可少的，另外，你应该会在 `usbcore` 的最后一行看到 `ehci_hcd` 或 `uhci_hcd`，它们就是前面说的 USB 主机控制器的驱动模块，你的 USB 设备要工作，合适的 USB 主机控制器模块也是必不可少的。

USB Core 负责实现一些核心的功能，为别的设备驱动程序提供服务，提供一个用于访问和控制 USB 硬件的接口，而不用去考虑系统当前存在哪种主机控制器。至于 USB Core、USB 主机控制器和 USB 设备驱动三者之间的关系，如图 1.9.1 所示。

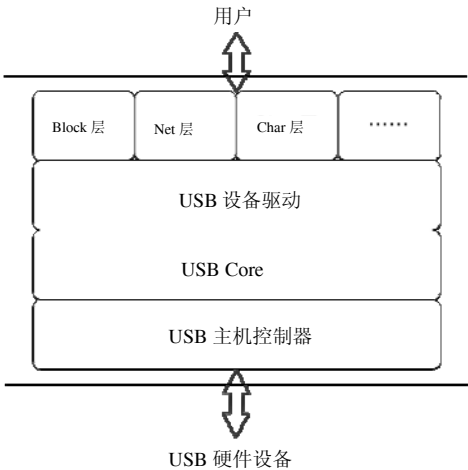


图 1.9.1 内核中 USB 子系统的结构

驱动和主机控制器像不像 `core` 的两个保镖？没办法，这可是 `core` 啊！协议中也说了，主机控制器的驱动（HCD）必须位于 USB 软件的最下一层。HCD 提供主机控制器硬件的抽象，隐藏硬件的细节，在主机控制器之下是物理的 USB 及所有与之连接的 USB 设备。而 HCD 只有一个客户，对一个人负责，就是咱们的 USB Core，USB Core 将用户的请求映射到相关的 HCD，用户不能直接访问 HCD。

在写 USB 驱动时，只能调用 `core` 的接口，`core` 会将咱们的请求发送给相应的 HCD，`core` 为咱们完成了大部分的工作，Linux 的哲学是不是和咱们生活中不太一样？

走到 `drivers/usb/core` 目录里去，使用 `ls` 命令看一看，

```
Kconfig Makefile buffer.c config.c devices.c devio.c driver.c
endpoint.c file.c generic.c hcd-pci.c hcd.c hcd.h hub.c hub.h
inode.c message.c notify.c otg_whitelist.h quirks.c sysfs.c urb.c
usb.c usb.h
```

再使用 `wc` 命令统计一下，将近两万行的代码，`core` 不愧是 `core`，为大家默默地做这么多事，我们要用感恩的心去深刻理解你的内心，回报你的付出。

不过这么多文件中不一定是我们所要关注的，先拿咱们的地图来看一看接下来该怎么走。先看一看 `Kconfig` 文件。

```
4 config USB_DEBUG
5     bool "USB verbose debug messages"
6     depends on USB
7     help
8     Say Y here if you want the USB core & hub drivers to produce a bunch
9     of debug messages to the system log. Select this if you are having a
10    problem with USB support and want to see more of what is going on.
```

这是 USB 的调试 `tag`，如果你在写 USB 设备驱动的话，最好还是打开它吧，不过这里它就不是我们关注的重点了。

```
15 config USB_DEVICEFS
16     bool "USB device filesystem"
17     depends on USB
18     ---help---
19     If you say Y here (and to "/proc file system support" in the "File
20     system s" section, above), you will get a file /proc/bus/usb/devices
21     which lists the devices currently connected to your USB bus or
22     busses, and for every connected device a file named
23     "/proc/bus/usb/xxx/yyy", where xxx is the bus number and yyy the
24     device number; the latter files can be used by user space programs
25     to talk directly to the device. These files are "virtual", meaning
26     they are generated on the fly and not stored on the hard drive.
27
28     You may need to mount the usbfs file system to see the files, use
29     mount -t usbfs none /proc/bus/usb
30
31     For the format of the various /proc/bus/usb/ files, please read
32     <file:Documentation/usb/proc_usb_info.txt>.
33
34     Usbfs files can't handle Access Control Lists (ACL), which are the
35     default way to grant access to USB devices for untrusted users of a
36     desktop system. The usbfs functionality is replaced by real
37     device-nodes managed by udev. These nodes live in /dev/bus/usb and
38     are used by libusb.
```

这个选项是关于 `usbfs` 文件系统的。`usbfs` 文件系统挂载在 `/proc/bus/usb` 上 (`mount -t usbfs none /proc/bus/usb`)，显示了当前连接的 USB 设备及总线的各种信息，每个连接的 USB 设备在其中都会有一个文件进行描述。比如文件 `/proc/bus/usb/xxx/yyy`，`xxx` 表示总线的序号，`yyy` 表示设备在总线的地址，不过不能够依赖它们来稳定地访问设备，因为同一个设备两次连接对应的

描述文件可能会不同。比如，第一次连接一个设备时，它可能是 002/027，一段时间后再次连接，它可能就已经改变为 002/048。

usbfs 与咱们探讨的主题关系不大，况且也已经足可以开个专题来讨论了，所以以后不会去过多提及它。

```
74 config USB_SUSPEND
75     bool "USB selective suspend/resume and wakeup (EXPERIMENTAL)"
76     depends on USB && PM && EXPERIMENTAL
77     help
78         If you say Y here, you can use driver calls or the sysfs
79         "power/state" file to suspend or resume individual USB
80         peripherals.
81
82         Also, USB "remote wakeup" signaling is supported, whereby some
83         USB devices (like keyboards and network adapters) can wake up
84         their parent hub. That wakeup cascades up the USB tree, and
85         could wake the system from states like suspend-to-RAM.
86
87         If you are unsure about this, say N here.
```

这一项是有关 USB 设备的挂起和恢复。开发 USB 的人都是节电节能的好孩子，所以协议中就规定了：所有的设备都必须支持挂起状态，也就是说为了达到节电的目的，当设备在指定的时间内（3 ms），如果没有发生总线传输，就要进入挂起状态。当它收到一个 non-idle 的信号时，就会被唤醒。节约用电从 USB 做起。不过目前来说内核对挂起休眠的支持普遍都不太好，而且许多的 USB 设备也没有支持它，还是暂且不提了。

剩下的还有几项，不过似乎与咱们关系也不大，还是去看一看 Makefile。

```
5 usbcore-objs      := usb.o hub.o hcd.o urb.o message.o driver.o \
6                    config.o file.o buffer.o sysfs.o endpoint.o \
7                    devio.o notify.o generic.o quirks.o
8
9 ifeq ($(CONFIG_PCI),y)
10     usbcore-objs    += hcd-pci.o
11 endif
12
13 ifeq ($(CONFIG_USB_DEVICEFS),y)
14     usbcore-objs    += inode.o devices.o
15 endif
16
17 obj-$(CONFIG_USB)  += usbcore.o
18
19 ifeq ($(CONFIG_USB_DEBUG),y)
20 EXTRA_CFLAGS += -DDEBUG
21 endif
```

Makefile 可比 Kconfig 简略多了，所以看起来也更亲切点，咱们总是拿的钱越多越好，看的代码越少越好。这里之所以会出现 CONFIG\_PCI，是因为通常 USB 的 Root Hub 包含在一个 PCI 设备中，前面也已经聊过了。hcd-pci 和 hcd 顾名思义就是主机控制器，它们实现了主机控制器公共部分，按协议中的说法它们就是 HCDI（HCD 的公共接口），host 目录下则实现了各

种不同的主机控制器。CONFIG\_USB\_DEVICEFS 在前面的 Kconfig 文件中也见到了,关于 usbfs 与咱们的主题无关,inode.c 和 devices.c 两个文件也可以不用管了。

## 10. 从这里开始

USB Core 从 USB 子系统的初始化开始,我们也需要从那里开始,它们位于文件 drivers/usb/core/usb.c 中:

```
938 subsys_initcall(usb_init);
939 module_exit(usb_exit);
```

我们看到一个 subsys\_initcall,它也是一个宏,我们可以把它理解为 module\_init,只不过因为这部分代码比较核心,开发人员们把它看做一个子系统,而不仅仅是一个模块。这也很好理解,usbcore 这个模块代表的不是某一个设备,而是所有 USB 设备赖以生存的模块,在 Linux 中,像这样一个类别的设备驱动被归结为一个子系统。比如 PCI 子系统,比如 SCSI 子系统,基本上,drivers/目录下面第一层的每个目录都算一个子系统,因为它们代表了一类设备。

subsys\_initcall(usb\_init)的意思就是告诉我们,usb\_init 是 USB 子系统真正的初始化函数,而 usb\_exit()将是整个 USB 子系统的结束时的清理函数,于是我们就从 usb\_init 开始看起。

```
863 static int __init usb_init(void)
864 {
865     int retval;
866     if (!nouseb) {
867         pr_info("%s: USB support disabled\n", usbcore_name);
868         return 0;
869     }
870
871     retval = ksuspend_usb_init();
872     if (retval)
873         goto out;
874     retval = bus_register(&usb_bus_type);
875     if (retval)
876         goto bus_register_failed;
877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;
880     retval = usb_major_init();
881     if (retval)
882         goto major_init_failed;
883     retval = usb_register(&usbfs_driver);
884     if (retval)
885         goto driver_register_failed;
886     retval = usb_devio_init();
887     if (retval)
888         goto usb_devio_init_failed;
889     retval = usbfs_init();
```

```
890     if (retval)
891         goto fs_init_failed;
892     retval = usb_hub_init();
893     if (retval)
894         goto hub_init_failed;
895     retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
896     if (!retval)
897         goto out;
898
899     usb_hub_cleanup();
900 hub_init_failed:
901     usbfs_cleanup();
902 fs_init_failed:
903     usb_devio_cleanup();
904 usb_devio_init_failed:
905     usb_deregister(&usbfs_driver);
906 driver_register_failed:
907     usb_major_cleanup();
908 major_init_failed:
909     usb_host_cleanup();
910 host_init_failed:
911     bus_unregister(&usb_bus_type);
912 bus_register_failed:
913     ksuspend_usb_cleanup();
914 out:
915     return retval;
916 }
```

首先看 863 行的 `__init` 标记，写过驱动的人应该不会陌生，它对内核来说就是一种暗示，表明这个函数仅在初始化期间使用，在模块被装载之后，它占用的资源就会释放掉用作它处。它的暗示你懂，可你的暗示，它却不懂或者懂装不懂，多么让人感伤。它在自己短暂的一生中一直从事繁重的工作，吃的是草吐出的是牛奶，留下的是整个 USB 子系统的繁荣。

受这种精神所感染，我觉得还是有必要为它说得更多一些。`__init` 的定义在 `include/linux/init.h` 文件中：

```
43 #define __init      __attribute__((__section__ (".init.text")))
```

好像这里引出了更多的疑问，`__attribute__` 是什么？Linux 内核代码使用了大量的 GNU C 扩展，以至于 GNU C 成为能够编译内核的唯一编译器，GNU C 的这些扩展对代码优化、目标代码布局、安全检查等方面也提供了很强的支持。而 `__attribute__` 就是这些扩展中的一个，它主要被用来声明一些特殊的属性，这些属性主要被用来指示编译器进行特定方面的优化和更仔细的代码检查。GNU C 支持十几个属性，`section` 是其中的一个，我们查看 GCC 的手册可以看到下面的描述：

```
'section ("section-name")'
Normally, the compiler places the code it generates in the `text'
section. Sometimes, however, you need additional sections, or you
need certain particular functions to appear in special sections.
The `section' attribute specifies that a function lives in a
particular section. For example, the declaration:
```

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function 'foobar' in the 'bar' section.

Some file formats do not support arbitrary sections so the 'section' attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

通常编译器将函数放在.text节，变量放在.data节或.bss节，使用section属性，可以让编译器将函数或变量放在指定的节中。那么前面对\_\_init的定义便表示将它修饰的代码放在.init.text节。连接器可以把相同节的代码或数据安排在一起，比如\_\_init修饰的所有代码都会被放在.init.text节里，初始化结束后就可以释放这部分内存。

那内核又是如何调用到这些\_\_init修饰的初始化函数？要回答这个问题，还需要回顾一下第938行的代码，上面已经提到subsys\_initcall也是一个宏，它也在include/linux/init.h中定义：

```
125 #define subsys_initcall(fn)                __define_initcall("4",fn,4)
```

这里又出现了一个宏\_\_define\_initcall，它用于将指定的函数指针fn放到initcall.init节里，而对于具体的subsys\_initcall宏，则是把fn放到.initcall.init的子节.initcall4.init里。要弄清楚.initcall.init、.init.text和.initcall4.init，我们还需要了解一点内核可执行文件相关的概念。

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init数据、bss等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。vmlinux.lds是存在于arch/<target>/目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。

我可以负责任地告诉你，要看懂vmlinux.lds这个文件是需要花一番工夫的，不过大家都是聪明人，聪明人做聪明事，所以你需要做的只是搜索initcall.init，然后便会看到似曾相识的内容。

```
__initcall_start = .;
.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {
    *(.initcall1.init)
    *(.initcall2.init)
    *(.initcall3.init)
    *(.initcall4.init)
    *(.initcall5.init)
    *(.initcall6.init)
    *(.initcall7.init)
}
__initcall_end = .;
```

这里的\_\_initcall\_start指向.initcall.init节的开始，\_\_initcall\_end指向它的结尾。而.initcall.init

节又被分为了 7 个子节，分别如下所示。

```
.initcall11.init
.initcall12.init
.initcall13.init
.initcall14.init
.initcall15.init
.initcall16.init
.initcall17.init
```

我们的 `subsys_initcall` 宏便是将指定的函数指针放在了 `.initcall14.init` 子节。其他的比如 `core_initcall` 将函数指针放在 `.initcall11.init` 子节，`device_initcall` 将函数指针放在了 `.initcall16.init` 子节等，都可以从 `include/linux/init.h` 文件找到它们的定义。各个字节的顺序是确定的，即先调用 `.initcall11.init` 中的函数指针再调用 `.initcall12.init` 中的函数指针等。`__init` 修饰的初始化函数在内核初始化过程中调用的顺序和 `.initcall.init` 节里函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此也就决定了它们的调用顺序。

至于实际执行函数调用的地方，就在 `/init/main.c` 文件中，内核的初始化不在那里还能在哪里？`do_initcalls` 函数会直接用到这里的 `__initcall_start`、`__initcall_end` 来进行判断。不多说了，还是回到久违的 `usb_init` 函数吧。

---

## 11. 面纱

前面说了那么多，才接触到 `usb_init`。当然，我们并不需要去经历爱情、被判与死亡，所需要经历的只是忍受前面大段大段的唠叨。

因为被 `__init` 给盯上，`usb_init` 在做牛做马的辛勤劳作之后便不得不灰飞烟灭，不可谓不高尚，但它始终只能是我们了解面纱后面内容的跳板，是起点，却不是终点，我们不会为它停留太久，有太多的精彩和苦恼在等着我们。

```
866     if (nousb) {
867         pr_info("%s: USB support disabled\n", usbcore_name);
868         return 0;
869     }
```

866 行，知道 C 语言的人都会知道 `nousb` 是一个标志，只是不同的标志有不一样的精彩。这里的 `nousb` 是用来让我们在启动内核时通过内核参数去掉 USB 子系统的，Linux 社会是一个很人性化的世界，它不会去逼迫我们接受 USB，一切都只关乎我们自己的需要。不过我想我们一般来说是不会去指定 `nousb` 的吧，毕竟它那么的讨人喜欢。如果你真的指定了 `nousb`，那它就会幽怨地说一句“USB support disabled”，然后退出 `usb_init`。

867 行，`pr_info` 只是一个打印信息的可辨参数宏，`printk` 的变体，在 `include/linux/kernel.h`

中定义：

```
242 #define pr_info(fmt,arg...) \
243     printk(KERN_INFO fmt,##arg)
```

1999 年的 ISO C 标准里规定了可变参数宏，与函数语法类似，比如：

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

里面的“...”就表示可变参数，调用时，它们就会替代宏体里的\_\_VA\_ARGS\_\_。GCC 总是会显得特立独行一些，它支持更复杂的形式，可以给可变参数取个名字，比如：

```
#define debug(format, args...) fprintf (stderr, format, args)
```

有了名字总是会容易交流一些。是不是与 pr\_info 比较接近了？除了“##”，它主要是针对空参数的情况。既然说是可变参数，那传递空参数也总是可以的。如果没有“##”，传递空参数时，比如：

```
debug ("A message");
```

展开后，里面的字符串后面会多一个多余的逗号。这个逗号你应该不会喜欢，而“##”则会使预处理器去掉这个多余的逗号。

```
871     retval = ksuspend_usb_init();
872     if (retval)
873         goto out;
874     retval = bus_register(&usb_bus_type);
875     if (retval)
876         goto bus_register_failed;
877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;
880     retval = usb_major_init();
881     if (retval)
882         goto major_init_failed;
883     retval = usb_register(&usbfs_driver);
884     if (retval)
885         goto driver_register_failed;
886     retval = usb_devio_init();
887     if (retval)
888         goto usb_devio_init_failed;
889     retval = usbfs_init();
890     if (retval)
891         goto fs_init_failed;
892     retval = usb_hub_init();
893     if (retval)
894         goto hub_init_failed;
895     retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
896     if (!retval)
897         goto out;
```

871 行到 897 行是代码里的排比句，相似的 init，不相似的内容，很显然都是在完成一些初始化，也是 usb\_init 任劳任怨所付出的全部。这里先简单地说一下。



871 行，电源管理方面的。如果在编译内核时没有打开电源管理，也就是说没有定义 `CONFIG_PM`，它就什么也不做。

874 行，注册 USB 总线，只有成功地将 USB 总线子系统注册到系统中，我们才可以向这个总线添加 USB 设备。

877 行，执行主机控制器相关的初始化。

880 行，一个总线同时也是一个设备，必须单独注册，因为 USB 是通过快速串行通信来读写数据，这里把它当做了字符设备来注册。

883 行~891 行，都是 `usbfs` 相关的初始化。

892 行，Hub 的初始化。

895 行，注册 USB 设备驱动，看清楚了，是 `USB device driver` 而不是 `USB driver`。前面说过，一个设备可以有多个接口，每个接口对应不同的驱动程序，这里所谓的 `device driver` 对应的是整个设备，而不是某个接口。内核中结构到处有，只是 USB 在这里格外多。

剩下的几行代码都是有关资源清除的，`usb_init` 这个短短的函数在承载着我们的希望时戛然而止了，你的感觉是什么？我的感觉是：这哪是我能说得清楚的啊！它的每个分叉都更像是一个陷阱，黑黝黝的看不到底，但是已经没有回头的路。

---

## 12. 模型，又见模型

上文说 `usb_init` 给我们留下了一些岔路口，每条都像是不归路，让人不知道从何处开始，也看不到路的尽头。趁着徘徊彷徨时，咱们还是先聊一下 Linux 的设备模型。

顾名思义，设备模型是关于设备的模型，对咱们写驱动的和不用写驱动的人来说，设备的概念就是总线与与其相连的各种设备了。电脑城的 IT 工作者都会知道设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是总线是如何发现设备的？设备又是如何和驱动对应起来的？它们经过怎样的艰辛才找到命里注定的那个它？它们的关系如何？白头偕老型的还是朝三暮四型的？这些问题就不是他们关心的了，而是咱们需要关心的。我们惊喜地发现，这些疑问的中心思想中心词汇就是总线、设备和驱动，没错，它们都是咱们这里要聊的 Linux 设备模型的“名角”。

总线、设备、驱动，也就是 `bus`、`device`、`driver`，既然是“名角”，在内核中都会有它们自己专属的结构，在 `include/linux/device.h` 中定义：

```

52 struct bus_type {
53     const char          * name;
54     struct module       * owner;
55
56     struct kset          subsys;
57     struct kset          drivers;
58     struct kset          devices;
59     struct klist         klist_devices;
60     struct klist         klist_drivers;
61
62     struct blocking_notifier_head bus_notifier;
63
64     struct bus_attribute * bus_attrs;
65     struct device_attribute * dev_attrs;
66     struct driver_attribute * drv_attrs;
67     struct bus_attribute drivers_autoprobe_attr;
68     struct bus_attribute drivers_probe_attr;
69
70     int (*match)(struct device * dev, struct device_driver * drv);
71     int (*uevent)(struct device *dev, char **envp,
72     int num_envp, char *buffer, int buffer_size);
73     int (*probe)(struct device * dev);
74     int (*remove)(struct device * dev);
75     void (*shutdown)(struct device * dev);
76
77     int (*suspend)(struct device * dev, pm_message_t state);
78     int (*suspend_late)(struct device * dev, pm_message_t state);
79     int (*resume_early)(struct device * dev);
80     int (*resume)(struct device * dev);
81
82     unsigned int drivers_autoprobe:1;
83 };

```

```

124 struct device_driver {
125     const char          * name;
126     struct bus_type     * bus;
127
128     struct kobject       kobj;
129     struct klist         klist_devices;
130     struct klist_node    knode_bus;
131
132     struct module       * owner;
133     const char          * mod_name; /* used for built-in modules */
134     struct module_kobject * mkobj;
135
136     int (*probe)         (struct device * dev);
137     int (*remove)        (struct device * dev);
138     void (*shutdown)     (struct device * dev);
139     int (*suspend)       (struct device * dev, pm_message_t state);
140     int (*resume)        (struct device * dev);
141 };

```

```

410 struct device {
411     struct klist         klist_children;
412     struct klist_node    knode_parent; /* node in sibling list */
413     struct klist_node    knode_driver;
414     struct klist_node    knode_bus;
415     struct device        *parent;
416

```

```

417     struct kobject kobj;
418     char    bus_id[BUS_ID_SIZE]; /* position on parent bus */
419     struct device_type *type;
420     unsigned    is_registered:1;
421     unsigned    uevent_suppress:1;
422     struct device_attribute uevent_attr;
423     struct device_attribute *devt_attr;
424
425     struct semaphore    sem; /* semaphore to synchronize calls to
426                             * its driver.
427                             */
428
429     struct bus_type * bus; /* type of bus device is on */
430     struct device_driver *driver; /* which driver has allocated this
431                                 device */
432     void    *driver_data; /* data private to the driver */
433     void    *platform_data; /* Platform specific data, device
434                             core doesn't touch it */
435     struct dev_pm_info    power;
436
437 #ifdef CONFIG_NUMA
438     int    numa_node; /* NUMA node this device is close to */
439 #endif
440     u64    *dma_mask; /* dma mask (if dma'able device) */
441     u64    coherent_dma_mask; /* Like dma_mask, but for
442                             alloc_coherent mappings as
443                             not all hardware supports
444                             64 bit addresses for consistent
445                             allocations such descriptors. */
446
447     struct list_head    dma_pools; /* dma pools (if dma'ble) */
448
449     struct dma_coherent_mem *dma_mem; /* internal for coherent mem
450                                     override */
451     /* arch specific additions */
452     struct dev_archdata    archdata;
453
454     spinlock_t    devres_lock;
455     struct list_head    devres_head;
456
457     /* class_device migration path */
458     struct list_head    node;
459     struct class    *class;
460     dev_t    devt; /* dev_t, creates the sysfs "dev" */
461     struct attribute_group **groups; /* optional groups */
462
463     void    (*release)(struct device * dev);
464 };

```

有没有发现它们的共性是什么？对，它们很长，很复杂。不过不妨把它们看成艺术品，既然是艺术品，当然不会让你那么容易就看懂了。

让我们平心静气地看一下上面代码的结构，我们会发现，struct bus\_type 中有成员 struct kset drivers 和 struct kset devices，同时 struct device 中有两个成员 struct bus\_type \* bus 和 struct device\_driver \*driver，struct device\_driver 中有两个成员 struct bus\_type \* bus 和 struct klist

klist\_devices。先不说什么是 klist、kset，光从成员的名字看，它们就是一个完美的三角关系。我们每个人心中是不是都有两个她？一个梦中的她，一个现实中的她。

我们可以知道 struct device 中的 bus 表示这个设备连到哪个总线上，driver 表示这个设备的驱动是什么。struct device\_driver 中的 bus 表示这个驱动属于哪个总线，klist\_devices 表示这个驱动都支持哪些设备，因为这里 device 是复数，又是 list，更因为一个驱动可以支持多个设备，而一个设备只能绑定一个驱动。当然，struct bus\_type 中的 drivers 和 devices 分别表示了这个总线拥有哪些设备和哪些驱动。

我们还需要看一看什么是 klist 和 kset。还有上面 device 和 driver 结构中出现的 kobject 结构是什么？我可以肯定地告诉你，kobject 和 kset 都是 Linux 设备模型中最基本的元素，总线、设备、驱动是西瓜，kobject、klist 是种瓜的人，没有幕后种瓜人的汗水不会有清爽解渴的西瓜。我们不能光知道西瓜是多么的甜，还要知道种瓜人的辛苦。kobject 和 kset 不会在意自己的得失，它们存在的意义在于把总线、设备和驱动这样的对象连接到设备模型上。种瓜的人也不会在意自己的汗水，在意的只是能不能种出甜蜜的西瓜。

一般来说应该这么理解，整个 Linux 的设备模型是一个 OO 的体系结构，总线、设备和驱动都是其中鲜活存在的对象，kobject 是它们的基类，所实现的只是一些公共的接口，kset 是同种类型 kobject 对象的集合，也可以说是对象的容器。只是因为 C 语言里不可能会有 C++ 语言里类的 class 继承、组合等的概念，只有通过 kobject 嵌入到对象结构中来实现。这样，内核使用 kobject 将各个对象连接起来组成了一个分层的结构体系。kobject 结构中包含了 parent 成员，指向了另一个 kobject 结构，也就是这个分层结构的上一层结点。而 kset 是通过链表来实现的，这样就可以明白，struct bus\_type 结构中的成员 drivers 和 devices 表示了一条总线拥有两条链表，一条是设备链表，一条是驱动链表。我们知道了总线对应的数据结构，就可以找到这条总线关联了多少设备，又有哪些驱动来支持这类设备。

那么 klist 呢？其实它就包含了一个链表和一个自旋锁，我们暂且把它看成链表也无妨。本来在 2.6.11 内核中，struct device\_driver 结构的 devices 成员就是一个链表类型。

那么总线、设备和驱动之间是如何和谐共处的呢？先说一说总线中的那两条链表是怎么形成的。内核要求每次出现一个设备就要向总线汇报，或者说注册，每次出现一个驱动，也要向总线汇报，或者说注册。比如系统初始化时，会扫描连接了哪些设备，并为每一个设备建立起一个 struct device 的变量，每一次有一个驱动程序，就要准备一个 struct device\_driver 结构的变量。把这些变量统统加入相应的链表，device 插入 devices 链表，driver 插入 drivers 链表。这样通过总线就能找到每一个设备，每一个驱动。然而，假如计算机里只有设备却没有对应的驱动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。在它们遇见彼此之前，双方都如同路埂里的野草，一个飘啊飘，一个摇啊摇，谁也不知道未来在哪里，只能在生命的风里飘摇。于是总线上的两张表里就慢慢地就挂上了许多孤单的灵魂。devices 开始多了，drivers 开始多了，它们像是来自两个不同的世界，devices 们彼此取暖，drivers 们一起

狂欢，但它们有一点是相同的，都只是在等待属于自己的另一半。

现在，总线上的两条链表已经有了，剩下的那个呢？链表里的设备和驱动又是如何联系的？先有设备还是先有驱动？很久很久以前，先有的是设备，每一个要用的设备在计算机启动之前就已经插好了，插放在它应该在的位置上，然后计算机启动，操作系统开始初始化，总线开始扫描设备，每找到一个设备，就为其申请一个 `struct device` 结构，并且挂入总线中的 `devices` 链表中来。然后每一个驱动程序开始初始化，开始注册其 `struct device_driver` 结构，然后去总线的 `devices` 链表中去寻找（遍历），去寻找每一个还没有绑定驱动的设备，即 `struct device` 中的 `struct device_driver` 指针仍为空的设备，然后它会去观察这种设备的特征，看是否是它所支持的设备，如果是，那么调用一个叫做 `device_bind_driver` 的函数，然后它们就结为了“秦晋之好”。换句话说，把 `struct device` 中的 `struct device_driver driver` 指向这个驱动，而 `struct device_driver driver` 把 `struct device` 加入它的那张 `struct klist klist_devices` 链表中来。就这样，`bus`、`device` 和 `driver`，这三者之间或者说他们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个。一荣俱荣，一损俱损。

但现在情况变了，出现了一种新的名词，叫热插拔。此时设备可以在计算机启动以后再插入或者拔出计算机了。因此，很难再说是先有设备还是先有驱动了，因为都有可能。设备可以在任何时刻出现，而驱动也可以在任何时刻被加载，所以，现在的情况就是，每当一个 `struct device` 诞生，它就会去 `bus` 的 `drivers` 链表中寻找自己的另一半。反之，每当一个 `struct device_driver` 诞生，它就去 `bus` 的 `devices` 链表中寻找它的那些设备。如果找到了合适的，那么和之前那种情况一样，调用 `device_bind_driver` 绑定好。如果找不到，没有关系，就等待吧。

---

## 13. 繁华落尽

看过了 Linux 设备模型固的繁花似锦，该是体味境界之美了。

Linux 设备模型中的总线落实在 USB 子系统里就是 `usb_bus_type`，它在 `usb_init` 函数的 874 行注册，在 `drivers/usb/core/driver.c` 文件中定义：

```
1523 struct bus_type usb_bus_type = {
1524     .name = "usb",
1525     .match = usb_device_match,
1526     .uevent = usb_uevent,
1527     .suspend = usb_suspend,
1528     .resume = usb_resume,
1529 };
```

`name` 自然就是 USB 总线的绰号。`match` 这个函数指针就比较有意思了，它充当了一个红娘的角色，在总线的设备和驱动之间“牵线搭桥”，但明显这里 `match` 的条件不是那么苛刻，要

更为实际一些。match 指向了函数 usb\_device\_match。

```

540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545         /* interface drivers never match devices */
546         if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551     } else {
552         struct usb_interface *intf;
553         struct usb_driver *usb_drv;
554         const struct usb_device_id *id;
555
556         /* device drivers never match interfaces */
557         if (is_usb_device_driver(drv))
558             return 0;
559
560         intf = to_usb_interface(dev);
561         usb_drv = to_usb_driver(drv);
562
563         id = usb_match_id(intf, usb_drv->id_table);
564         if (id)
565             return 1;
566
567         id = usb_match_dynamic_id(intf, usb_drv);
568         if (id)
569             return 1;
570     }
571 }
572
573 return 0;
574 }

```

540 行，经历了繁华的 Linux 设备模型，这两个参数我们都已经很熟悉了，对应的就是总线两条链表里的设备和驱动。总线上有新设备或新的驱动添加时，这个函数总是会被调用，如果指定的驱动程序能够处理指定的设备，也就是匹配成功，函数返回 0。梦想是美好的，现实是残酷的，匹配是未必会成功的，红娘再努力，双方对不上眼也是实在没办法的事。

543 行，一遇到 if 和 else，我们就知道又处在两难境地了，代码里我们可选择的太多，生活里我们可选择的太少。这里的岔路口只有两条路：一条给 USB 设备走，一条给 USB 接口走，各走各的路，分开了，就不再相见。

## 14. 接口是设备的接口

设备可以有多个接口，每个接口代表一个功能，每个接口对应着一个驱动。Linux 设备模型中的 device 落在 USB 子系统，成了两个结构，一个是 struct usb\_device，一个是 struct usb\_interface。一个 USB 键盘，上面带一个扬声器，因此有两个接口，那肯定得要两个驱动程序，一个是键盘驱动程序，一个是音频流驱动程序。“道”上的兄弟喜欢把这样两个整合在一起的东西叫做一个设备，那好，让他们去叫吧，我们用 interface 来区分这两者。于是有了这里提到的数据结构，struct usb\_interface：

```
140 struct usb_interface {
141     /* array of alternate settings for this interface,
142      * stored in no particular order */
143     struct usb_host_interface *altsetting;
144
145     struct usb_host_interface *cur_altsetting; /* the currently
146                                                  * active alternate setting */
147     unsigned num_altsetting; /* number of alternate settings */
148
149     int minor; /* minor number this interface is
150               * bound to */
151     enum usb_interface_condition condition; /* state of binding */
152     unsigned is_active:1; /* the interface is not suspended */
153     unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */
154
155     struct device dev; /* interface specific device info */
156     struct device *usb_dev; /* pointer to the usb class's device, if any */
157     int pm_usage_cnt; /* usage counter for autosuspend */
158 };
```

143 行，这里有一个 altsetting 成员，它的意思就是 alternate setting，可选的设置。那么知道 145 行的 cur\_altsetting 表示当前正在使用的设置，147 行的 num\_altsetting 表示这个接口具有可选设置的数量。前面提到过 USB 设备的配置，那这里的设置是什么意思？

在英语中，配置是 configuration，设置是 setting。

先说配置，一部手机可以有多种配置，比如可以摄像，还可以连接在计算机里当做一个 U 盘，那么这两种情况就属于不同的配置，在手机里面有相应的选择菜单，你选择了哪种它就按哪种配置进行工作，供你选择的这个选项就叫做配置。很显然，当你摄像时你不可以把手机当做 U 盘进行访问，当你把手机当做 U 盘进行访问时你不可以摄像。第二，既然一个配置代表一种不同的功能，那么很显然，不同的配置可能需要的接口就不一样，假设你的手机从硬件上来说一共有 5 个接口，那么可能当你配置成 U 盘时它只需要用到某一个接口；当你配置成摄像时，它可能只需要用到另外两个接口；也许你还有别的配置，可能就会用到剩下的那两个接口。

再说一说设置，一部手机可能各种配置都确定了，是振动还是铃声已经确定了，各种功能也都确定了，但是声音的大小还可以变吧，通常手机的音量是一格一格地变动，大概也就 5 格至 6 格，那么这个可以算一个设置。

如果你还是不明白什么是配置什么是设置的话，那就直接用它们大小关系来理解好了，毕竟大家对互相之间的大小关系都更敏感一些，不要说不是。这么说吧，设备大于配置，配置大于接口，接口大于设置。更准确地说是设备可以有多个配置，配置里可以包含一个或更多的接口，而接口通常又具有一个或更多的设置。

149 行，minor，分配给接口的次设备号。Linux 下所有的硬件设备都是用文件来表示的，俗称“设备文件”，在/dev 目录下面，为了显示自己并不是普通的文件，它们都会有一个主设备号和次设备号，如下所示：

```
brw-r----- 1 root disk 8, 0 Sep 26 09:17 /dev/sda
brw-r----- 1 root disk 8, 1 Sep 26 09:17 /dev/sda1
crw-r----- 1 root tty 4, 1 Sep 26 09:17 /dev/tty1
```

这是在/dev 目录下执行 ls 命令后的部分显示结果。我们可以看到在每一行的日期前面有两个逗号隔开的数字，对于普通文件而言，这个位置显示的是文件的长度。而对于设备文件，这里显示的两个数字表示了该设备的主设备号和次设备号。一般来说，主设备号表明了设备的种类，也表明了设备对应着哪个驱动程序，而次设备号则是因为一个驱动程序要支持多个设备而为了让驱动程序区分它们而设置的。也就是说，主设备号用来帮你找到对应的驱动程序，次设备号决定你的驱动对哪个设备进行操作。在上面代码中就显示了我的移动硬盘主设备号为 8，系统里 tty 设备的主设备号为 4。

设备要想在 Linux 里分得一个主设备号，有一个立足之地，也并不是那么容易的。主设备号虽说不是什么特别稀缺的资源，但还是需要设备先在驱动里提出申请，获得系统的批准后才能拥有一个。因为一部分的主设备号已经被静态地预先指定给了许多常见的设备，申请时要避开它们。这些已经被分配掉的主设备号都列在 Documentation/devices.txt 文件中。当然，如果你是用动态分配的形式，就可以不去理会这些，直接让系统为你做主，替你选择一个即可。

很显然，USB 设备是很常见的，linux 理应为它预留了一个主设备号。下面看一看 include/linux/usb.h 文件。

```
7 #define USB_MAJOR 180
8 #define USB_DEVICE_MAJOR 189
```

苏格拉底说过，学得越多，知道得越多；知道得越多，发现需要知道得更多。当我们知道了主设备号，满怀激情与向往地来寻找 USB 的主设备号时，我们却发现这里在上演真假李逵。这两个设备号哪个才是我们苦苦追寻的“她”？

你可以在内核中搜索它们都曾经出现什么地方，或者就跟随我回到 usb\_init 函数。

```
880     retval = usb_major_init();
881     if (retval)
882         goto major_init_failed;
883     retval = usb_register(&usbfs_driver);
884     if (retval)
885         goto driver_register_failed;
```



```
886     retval = usb_devio_init();
887     if (retval)
888         goto usb_devio_init_failed;
889     retval = usbfs_init();
890     if (retval)
891         goto fs_init_failed;
```

前面只提了一句 883 行到 891 行是与 `usbfs` 相关的代码就简单略过了，这里稍微多说一点。`usbfs` 提供了在用户空间直接访问 USB 硬件设备的接口，但是世界上没有免费的午餐，它需要内核的大力支持，`usbfs_driver` 就是用来完成这个光荣任务的。我们可以去 `usb_devio_init` 函数中看一看，它在 `drivers/usb/devio.c` 文件中定义：

```
retval = register_chrdev_region(USB_DEVICE_DEV, USB_DEVICE_MAX,
                                "usb_device");
if (retval) {
    err("unable to register minors for usb_device");
    goto out;
}
```

`register_chrdev_region` 函数获得了设备 `usb_device` 对应的设备编号，设备 `usb_device` 对应的驱动当然就是 `usbfs_driver`，参数 `USB_DEVICE_DEV` 也在同一个文件中有定义。

```
#define USB_DEVICE_DEV        MKDEV(USB_DEVICE_MAJOR, 0)
```

终于再次见到了 `USB_DEVICE_MAJOR`，也终于明白它是为了 `usbfs` 而生，为了让广大人民群众能够在用户空间直接和 USB 设备通信而生。因此，它并不是我们所要寻找的。

那么答案很明显了，`USB_MAJOR` 就是我们苦苦追寻的那个“她”，就是 Linux 为 USB 设备预留的主设备号。事实上，前面 `usb_init` 函数的 880 行，`usb_major_init` 函数已经使用 `USB_MAJOR` 注册了一个字符设备，名字就叫 `USB`。我们可以在文件 `/proc/devices` 里看到它们。

```
localhost:/usr/src/linux/drivers/usb/core # cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 tty
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
14 sound
29 fb
116 alsa
128 ptm
136 pts
162 raw
180 usb
189 usb_device
```

`/proc/devices` 文件中显示了所有当前系统里已经分配出去的主设备号，当然上面只是列出了字符设备，块设备被有意地略过了。很明显，前面提到的 `usb_device` 和 `usb` 都在里面。

不过到这里还没讲完，USB 设备有很多种，并不是都会用到这个预留的主设备号。比如我的移动硬盘显示出来的主设备号就是 8，你的摄像头在 Linux 显示的主设备号也绝对不会是这里的 `USB_MAJOR`。坦白地说，大多数 USB 设备都会与 `input`、`video` 等子系统关联，并不单单只是作为 USB 设备而存在。如果 USB 设备没有与其他任何子系统关联，就需要在对应驱动的 `probe` 函数中使用 `usb_register_dev` 函数来注册并获得主设备号 `USB_MAJOR`，你可以在 `drivers/usb/misc` 目录下看到一些例子，`drivers/usb/usb-skeleton.c` 文件也属于这种。如果 USB 设备关联了其他子系统，则需要在对应驱动程序的 `probe` 函数中使用相应的注册函数，`USB_MAJOR` 也就用不着了。比如，USB 键盘关联了 `input` 子系统，驱动对应 `drivers/hid/usbhid` 目录下的 `usbkbd.c` 文件，在它的 `probe` 函数中可以看到使用了 `input_register_device` 来注册一个输入设备。

准确地说，这里的 USB 设备应该说成 USB 接口，毕竟一个 USB 接口才对应着一个 USB 驱动。当 USB 接口关联有其他子系统，也就是说不使用 `USB_MAJOR` 作为主设备号时，`struct usb_interface` 的字段 `minor` 可以忽略。`minor` 只在 `USB_MAJOR` 起作用时才起作用。

说完了设备号，回到 `struct usb_interface` 的 151 行，`condition` 字段表示接口和驱动的绑定状态，`enum usb_interface_condition` 类型在 `include/linux/usb.h` 中定义。

```
83 enum usb_interface_condition {
84     USB_INTERFACE_UNBOUND = 0,
85     USB_INTERFACE_BINDING,
86     USB_INTERFACE_BOUND,
87     USB_INTERFACE_UNBINDING,
88 };
```

前面介绍 Linux 设备模型时说了，设备和驱动是相生相依的关系，总线上的每个设备和驱动都在等待着命中的那个“她”，找到了，就会“执子之手与子偕老”；找不到，只能“孤苦伶仃”一个人了。`enum usb_interface_condition` 形象地描绘了这个过程中接口的各种心情，孤苦、期待、幸福、分开，人生又何尝不是如此？

152 行、153 行与 157 行都是关于挂起和唤醒的代码。协议中规定，所有的 USB 设备都必须支持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内（大约 3 ms），如果没有发生总线传输，就要进入挂起状态。当它收到一个 `non-idle` 的信号时，就会被唤醒。

152 行，`is_active` 表示接口是否处于挂起状态。153 行，`needs_remote_wakeup` 表示是否需要打开远程唤醒功能。远程唤醒功能允许挂起的设备给主机发信号，通知主机它将从挂起状态恢复，注意如果主机处于挂起状态，就会唤醒主机，不然主机仍然在休眠。协议中并没有要求 USB 设备一定要实现远程唤醒的功能，即使实现了，从主机这边也可以打开或关闭它。

157 行中的 `pm_usage_cnt`, `pm` 就是电源管理; `usage_cnt` 就是使用计数, 当它为 0 时, 接口允许 `autosuspend`。什么是 `autosuspend`? 用过笔记本电脑吧, 有时合上笔记本电脑后, 它会自动进入休眠, 这就叫 `autosuspend`。但不是每次都是这样的, 就像这里只有当 `pm_usage_cnt` 为 0 时才会允许接口 `autosuspend`。

接下来就剩下 155 行的 `struct device dev` 和 156 行的 `struct device *usb_dev`, 看到 `struct device`, 或许就会认为它们是 Linux 设备模型中的 `device` 嵌套在这里的对象。不过这么想是不准确的, 这两个成员里面只有 `dev` 才是模型中的 `device` 嵌套在这里的, `usb_dev` 则不是。当接口使用 `USB_MAJOR` 作为主设备号时, `usb_dev` 才会用到。找遍整个内核, 也只在 `usb_register_dev` 和 `usb_deregister_dev` 两个函数中能够看到它, `usb_dev` 指向的就是 `usb_register_dev` 函数中创建的 USB class device。

---

## 15. 设置是接口的设置

不过这里还是有一点小悬念的, 前面介绍 `struct usb_interface` 时, 表示接口设置的 `struct usb_host_interface` 就被有意无意地略过了, 现在看一看它的真面目, 同样在 `include/linux/usb.h` 文件中定义。

```
70 struct usb_host_interface {
71     struct usb_interface_descriptor desc;
72
73     /* array of desc.bNumEndpoint endpoints associated with this
74      * interface setting. these will be in no particular order.
75      */
76     struct usb_host_endpoint *endpoint;
77
78     char *string;           /* iInterface string, if present */
79     unsigned char *extra;   /* Extra descriptors */
80     int extralen;
81 };
```

71 行, `desc`, 接口描述符。什么是描述符? 我们的生活就是一个不断遇到人, 认识人的过程, 有些人注定只是擦肩而过, 有些人却深深地留在我们的心里, 比如 USB 的描述符。实际上, USB 的描述符是一个带有预定义格式的数据结构, 里面保存了 USB 设备的各种属性还有相关信息, 比如姓名、生产地等, 我们可以通过向设备请求获得它们的内容来深刻地了解感知一个 USB 设备。

USB 描述符主要有四种: 设备描述符、配置描述符、接口描述符和端点描述符。协议中规定一个 USB 设备是必须支持这四种描述符, 当然也有其他一些描述符来让设备可以显得个性一些, 但这四大描述符是一个都不能少的。

这些描述符放哪儿？当然是在设备中，就等着主机去“拿”。具体在哪儿？USB 设备中都会有一个叫 EEPROM 的东西，没错，就是放在那儿，它就是用来存储设备本身信息的。EEPROM 就是电可擦写可编程 ROM，它与 Flash 虽说都是要电擦除的，但它可以按字节擦除，Flash 只能一次擦除一个 block，所以如果要修改比较少的数据的话，使用它还是比较合适的。但是世界上没有完美的东西，EEPROM 成本相对 Flash 比较高，所以一般来说 USB 设备中只拿它来存储一些本身特有的信息，要想存储数据，还是使用 Flash 吧。

接口描述符，就是描述接口本身的信息的。一个接口可以有多个设置，使用不同的设置，描述接口的信息会有所不同，所以接口描述符并没有放在 struct usb\_interface 结构中，而是放在表示接口设置的 struct usb\_host\_interface 结构中。在 include/linux/usb/ch9.h 文件中定义。

```
294 /* USB_DT_INTERFACE: Interface descriptor */
295 struct usb_interface_descriptor {
296     __u8  bLength;
297     __u8  bDescriptorType;
298
299     __u8  bInterfaceNumber;
300     __u8  bAlternateSetting;
301     __u8  bNumEndpoints;
302     __u8  bInterfaceClass;
303     __u8  bInterfaceSubClass;
304     __u8  bInterfaceProtocol;
305     __u8  iInterface;
306 } __attribute__((packed));
```

又看到了 \_\_attribute\_\_，不过在这里改头换面成了 \_\_attribute\_\_((packed))，意思就是告诉编译器，这个结构的元素都是 1 字节对齐的，不要再添加填充位了。因为这个结构和 spec 里的 Table 9.12 是完全一致的，包括字段的长度。如果不给编译器这个暗示，编译器就会依据你的平台类型在结构的每个元素之间添加一定的填充位，如果你拿这个添加了填充位的结构去向设备请求描述符，你想一想会是什么结果。

296 行，bLength，描述符的字节长度。协议中规定，每个描述符必须以一个字节打头来表明描述符的长度。接口描述符的 bLength 应该是 9，没错，ch9.h 文件中紧挨着接口描述符的定义就定义了这个长度。

```
308 #define USB_DT_INTERFACE_SIZE 9
```

297 行，bDescriptorType，描述符的类型。各种描述符的类型都在 ch9.h 文件中有定义，对应 spec 中的 Table 9.5。对于接口描述符来说，值为 USB\_DT\_INTERFACE，也就是 0x04。

299 行，bInterfaceNumber，接口号。每个配置可以包含多个接口，这个值就是它们的索引值。

300 行，bAlternateSetting，接口使用的是哪个可选设置。协议中规定，接口默认使用的设置总为 0 号设置。

301 行, `bNumEndpoints`, 接口拥有的端点数量。这里并不包括端点 0, 端点 0 是所有的设备都必须提供的, 所以这里就没必要多此一举包括它了。

302 行, `bInterfaceClass`; 303 行, `bInterfaceSubClass`; 304 行, `bInterfaceProtocol`。这个世界上有许多 USB 设备, 它们各有各的特点, 为了区分它们, USB 规范, 或者说 USB 协议把 USB 设备分成了很多类, 然而每个类又分成子类。这很好理解, 我们的一个大学也是如此, 先是分成很多个学院, 然后每个学院又被分为很多个系, 可能每个系下边又分了各个专业。USB 协议也是这样的, 首先每个 Device 或 Interface 属于一个 Class, 然后 Class 下面又分了 SubClass, SubClass 下面又按各种设备所遵循的不同的通信协议继续细分。USB 协议中边为每一种 Class, 每一种 SubClass, 每一种 Protocol 定义一个数值, 比如 Mass Storage 的 Class 就是 0x08, Hub 的 Class 就是 0x09。

305 行, `iInterface`, 接口对应的字符串描述符的索引值。这里怎么又跳出来一个叫字符串描述符的东西? 你没看错我也没说错, 除了前面提到的四大描述符, 还有字符串描述符。不过四大描述符是每个设备必须支持的, 这个字符串描述符却是可有可无的, 有了你欢喜, 我也欢喜, 没有也不是什么问题。使用 `lsusb` 命令看一下。

```
localhost:/usr/src/linux/drivers/usb/core # lsusb
Bus 001 Device 013: ID 04b4:1081 Cypress Semiconductor Corp.
Bus 001 Device 001: ID 0000:0000
```

在上面代码的第 1 行里显示的是我手上的 Cypress USB 开发板, 看里面的 Cypress Semiconductor Corp., 它从哪里来? 是不是应该从设备中来? 设备的那几个标准描述符, 整个描述符的大小也不一定放得下这么一长串, 所以, 一些设备专门准备了一些字符串描述符(string descriptor), 就用来记这些长串的东西。字符串描述符主要就是提供一些设备接口相关的描述性信息, 比如厂商的名字, 产品序列号等。字符串描述符当然可以有多个, 这里的索引值就是用来区分它们的。

说过了接口描述符, 回到 `struct usb_host_interface` 的 76 行, `endpoint`, 一个数组, 表示这个设置所使用到端点。

78 行, `string`, 用来保存从设备中取出来的字符串描述符信息的, 既然字符串描述符可有可无, 这里的指针也有可能为空了。

79 行, `extra`; 80 行, `extralen`, 有关额外的描述符。除了前面提到的四大描述符及字符串描述符外, 还有为一组设备也就是一类设备定义的描述符, 和厂商为设备特别定义的描述符, `extra` 指的就是它们, `extralen` 表示它们的长度。

## 16. 端点

端点是 USB 数据传输的终点。看一看它在内核中的定义。

```
59 struct usb_host_endpoint {
60     struct usb_endpoint_descriptor desc;
61     struct list_head      urb_list;
62     void                  *hcpriv;
63     struct ep_device      *ep_dev;      /* For sysfs info */
64
65     unsigned char *extra; /* Extra descriptors */
66     int extralen;
67 };
```

60 行, desc, 端点描述符, 四大描述符的第二个隆重登场了。它也在 include/linux/usb/ch9.h 中定义。

```
312 /* USB_DT_ENDPOINT: Endpoint descriptor */
313 struct usb_endpoint_descriptor {
314     __u8  bLength;
315     __u8  bDescriptorType;
316
317     __u8  bEndpointAddress;
318     __u8  bmAttributes;
319     __le16 wMaxPacketSize;
320     __u8  bInterval;
321
322     /* NOTE: these two are _only_ in audio endpoints. */
323     /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324     __u8  bRefresh;
325     __u8  bSynchAddress;
326 } __attribute__((packed));
327
328 #define USB_DT_ENDPOINT_SIZE      7
329 #define USB_DT_ENDPOINT_AUDIO_SIZE 9 /* Audio extension */
```

这个结构与 spec 中的 Table 9.13 是一一对应的, 0 号端点仍然保持着它特殊的地位, 它没有自己的端点描述符。

314 行, bLength, 描述符的字节长度, 前面有 7 个字节, 后面又多了两个字节, 那是针对音频设备扩展的, 紧接着 struct usb\_host\_endpoint 定义的就是两个长度值的定义。

315 行, bDescriptorType, 描述符类型, 这里对应的端点就是 USB\_DT\_ENDPOINT, 0x05。

317 行, bEndpointAddress, 这个字段描述的信息挺多的, 比如这个端点是输入端点还是输出端点, 这个端点的地址, 以及这个端点的端点号。它的 bits 0~bits 3 表示的就是端点号, 你使用 0x0f 和它相与就可以得到端点号。不过, 开发内核的人想得都很周到, 定义好了一个掩码 USB\_ENDPOINT\_NUMBER\_MASK, 它的值就是 0x0f。当然, 这是为了让我们更容易去读代码, 也为了以后的扩展。另外, 它的 bit 8 表示方向, 输入还是输出, 同样有掩码 USB\_ENDPOINT\_DIR\_MASK, 值为 0x80, 将它和 bEndpointAddress 相与, 并结合 USB\_DIR\_IN

和 USB\_DIR\_OUT 作判断就可以得到端点的方向。

```
48 #define USB_DIR_OUT          0          /* to device */
49 #define USB_DIR_IN           0x80       /* to host */
```

318 行, bmAttributes, 属性, 总共 8 位, 其中 bit1 和 bit0 共同称为 Transfer Type, 即传输类型。00 表示控制传输, 01 表示等时传输, 10 表示批量传输, 11 表示中断传输。前面的端点号还有端点方向都有配对的掩码, 这里当然也有, 就在 struct usb\_endpoint\_descriptor 定义的下面。

```
338 #define USB_ENDPOINT_XFERTYPE_MASK    0x03    /* in bmAttributes */
339 #define USB_ENDPOINT_XFER_CONTROL      0
340 #define USB_ENDPOINT_XFER_ISOC         1
341 #define USB_ENDPOINT_XFER_BULK         2
342 #define USB_ENDPOINT_XFER_INT          3
```

319 行, wMaxPacketSize, 端点一次可以处理的最大字节数。比如有好几个工作等你去处理, 但你只能一个一个地分开做。端点也是, 如果你发送的数据量大于端点的这个值, 也会分成多次, 一次一次来传输。友情提醒一下, 这个字段对不同的传输类型也有不同的要求。

320 行, bInterval, USB 是轮询式总线, 这个值表达了端点一种美好的期待, 希望主机轮询自己的时间间隔, 但实际上批准不批准就是主机的事了。不同的传输类型 bInterval 也有不同的意义, 暂时就提这么一下, 碰到各个实际的传输类型了再去说它。

回到 struct usb\_host\_endpoint, 61 行, urb\_list, 端点要处理的 urb 队列。urb 是什么? 它可是 USB 通信的主角, 包含了执行 USB 传输所需要的所有信息, 你要想和你的 USB 通信, 就得创建一个 urb, 并且为它赋值, 交给 USB Core, 然后 USB Core 会找到合适的主机控制器, 从而进行具体的数据传输。设备中的每个端点都可以处理一个 urb 队列。当然, urb 是内核中对 USB 传输数据的封装也叫抽象, 协议中可不这么叫。

62 行, hcpriv, 这是提供给 HCD (Host Controller Driver) 用的。比如等时端点会在里边儿放一个 ehci\_iso\_stream。

63 行, ep\_dev, 这个字段是供 sysfs 用的。好奇的话可以去/sys 下看一看。

```
localhost:/usr/src/linux # ls /sys/bus/usb/devices/usb1/ep_00/
bEndpointAddress bmAttributes  direction  subsystem  wMaxpacketSize
bInterval      dev      intervaltype
bLength        device    power      uevent
```

ep\_00 端点目录下的这些文件从哪儿来的? 就是在 usb\_create\_ep\_files 函数中使用 ep\_dev 创建的。

65 行, extra; 66 行, extralen, 有关一些额外扩展的描述符的, 和 struct usb\_host\_interface 里差不多, 只是这里的是针对端点的, 如果你请求从设备中获得描述符信息, 它们会跟在标准的端点描述符后面返回给你。

## 17. 设备

struct usb\_device 结构冗长而又杂乱，但还是得说。

```

336 struct usb_device {
337     int          devnum;          /* Address on USB bus */
338     char          devpath [16]; /* Use in messages: /port/port/... */
339     enum usb_device_state state; /* configured, not attached, etc */
340     enum usb_device_speed speed; /* high/full/low (or error) */
341
342     struct usb_tt    *tt;          /* low/full speed dev, highspeed hub */
343     int              ttport;       /* device port on that tt hub */
344
345     unsigned int toggle[2]; /* one bit for each endpoint
346                             * ([0] = IN, [1] = OUT) */
347
348     struct usb_device *parent; /* our hub, unless we're the root */
349     struct usb_bus *bus;       /* Bus we're part of */
350     struct usb_host_endpoint ep0;
351
352     struct device dev;          /* Generic device interface */
353
354     struct usb_device_descriptor descriptor; /* Descriptor */
355     struct usb_host_config *config; /* All of the configs */
356
357     struct usb_host_config *actconfig; /* the active configuration */
358     struct usb_host_endpoint *ep_in[16];
359     struct usb_host_endpoint *ep_out[16];
360
361     char **rawdescriptors; /* Raw descriptors for each config */
362
363     unsigned short bus_mA; /* Current available from the bus */
364     u8 portnum; /* Parent port number (origin 1) */
365     u8 level; /* Number of USB hub ancestors */
366
367     unsigned discon_suspended:1; /* Disconnected while suspended */
368     unsigned have_langid:1; /* whether string_langid is valid */
369     int string_langid; /* language ID for strings */
370
371     /* static strings from the device */
372     char *product; /* iProduct string, if present */
373     char *manufacturer; /* iManufacturer string, if present */
374     char *serial; /* iSerialNumber string, if present */
375
376     struct list_head filelist;
377 #ifdef CONFIG_USB_DEVICE_CLASS
378     struct device *usb_classdev;
379 #endif
380 #ifdef CONFIG_USB_DEVICEFS
381     struct dentry *usbfs_dentry; /*usbfs dentry entry for the device*/
382 #endif
383     /*
384      * Child devices - these can be either new devices
385      * (if this is a hub device), or different instances
386      * of this same device.
387      *
388      * Each instance needs its own set of data structures.

```



```

389     */
390
391     int maxchild;                /* Number of ports if hub */
392     struct usb_device *children[USB_MAXCHILDREN];
393
394     int pm_usage_cnt;            /* usage counter for autosuspend */
395     u32 quirks;                 /* quirks of the whole device */
396
397 #ifdef CONFIG_PM
398     struct delayed_work autosuspend; /* for delayed autosuspends */
399     struct mutex pm_mutex;       /* protects PM operations */
400
401     unsigned long last_busy;     /* time of last use */
402     int autosuspend_delay;      /* in jiffies */
403
404     unsigned auto_pm:1;         /* autosuspend/resume in progress */
405     unsigned do_remote_wakeup:1; /* remote wakeup should be enabled */
406     unsigned autosuspend_disabled:1; /* autosuspend and autoresume */
407     unsigned autoresume_disabled:1; /* disabled by the user */
408 #endif
409 };

```

337 行，`devnum`，设备的地址。此地址非彼地址，和我们写程序时说的地址不一样。`devnum` 只是 USB 设备在一条 USB 总线上的编号。你的 USB 设备插到 Hub 上时，Hub 观察到这个变化，会在一个漫长而又曲折的处理过程中调用一个名叫 `choose_address` 的函数，为你的设备选择一个地址。

有人说我没有用 Hub，我的 USB 设备直接插到主机的 USB 接口上了。不管是一般的 Hub 还是 Root Hub，你的 USB 设备总要通过一个 Hub 才能在 USB 的世界里生活。

现在来认识一下 USB 子系统里面关于地址的游戏规则。在 USB 世界里，一条总线就是一棵树一棵，一个设备就是一片叶子。为了记录这棵树上的每一个叶子节点，每条总线设有一个地址映射表，即 `struct usb_bus` 结构体里有一个成员 `struct usb_devmap devmap`。

```

268 /* USB device number allocation bitmap */
269 struct usb_devmap {
270     unsigned long devicemap[128 / (8*sizeof(unsigned long))];
271 };

```

什么是 `usb_bus`？前面不是已经有一个 `struct bus_type` 类型的 `usb_bus_type` 了吗？没错，在 USB 子系统的初始化函数 `usb_init` 里已经注册了 `usb_bus_type`，不过那是让系统知道有这么一个类型的总线。而一个总线类型和一条总线是两码事儿。从硬件上来讲，一个主机控制器就会连出一条 USB 总线；而从软件上来讲，不管你有多少个主机控制器，或者说有多少条总线，它们通通属于 `usb_bus_type` 这个类型，只是每一条总线对应一个 `struct usb_bus` 结构体变量，这个变量在主机控制器的驱动程序中申请。

上面的 `devmap` 地址映射表就表示了一条总线上设备连接的情况，假设 `unsigned long` 为 4Bytes，那么 `unsigned long devicemap[128/(8*sizeof(unsigned long))]` 就等价于 `unsigned long devicemap[128/(8*4)]`，进而等价于 `unsigned long devicemap[4]`，而 4Bytes 就是 32 bits，因此这

个数组最终表示的就是 128 bits。而这也对应于一条总线可以连接 128 个 USB 设备。之所以这里使用 `sizeof(unsigned long)`，就是为了跨平台应用，不管 `unsigned long` 到底是几，总之这个 `devicemap` 数组最终可以表示 128 位，也就是说每条总线上最多可以连上 128 个设备。

338 行，`devpath [16]`，它显然是用来记录一个字符串的，这个字符串是什么意思？如下所示。

```
localhost:~ # ls /sys/bus/usb/devices/
1-0:1.0  2-1      2-1:1.1  4-0:1.0  4-5:1.0  usb2  usb4
2-0:1.0  2-1:1.0  3-0:1.0  4-5      usb1   usb3
```

在 `sysfs` 文件系统下，我们可以看到这些乱七八糟的东西，它们都是什么？`usb1`、`usb2`、`usb3`、`usb4` 表示我的计算机上接了 4 条 USB 总线，即 4 个 USB 主机控制器，事物多了自然就要编号，就跟我们在中学或大学里的学号一样，用于区分多个个体。

而 `4-0:1.0` 表示什么？4 表示是 4 号总线，或者说 4 号 Root Hub；0 就是这里我们说的 `devpath`，1 表示配置为 1 号，0 表示接口号为 0。也就是说，4 号总线的 0 号端口的设备，使用的是 1 号配置，接口号为 0。

那么 `devpath` 是否就是端口号呢？显然不是，这里我列出来的这个例子是只有 Root Hub 没有级联 Hub 的情况，如果在 Root Hub 上又接了别的 Hub，然后一级一级连下去。那么如何在 `sysfs` 里面来表示这整个大家族呢？这就是 `devpath` 的作用，顶级设备的 `devpath` 就是其连在 Root Hub 上的端口号，而次级的设备就是其父 hub 的 `devpath` 后面加上其端口号，即如果 `4-0:1.0` 如果是一个 Hub，那么它下面的 1 号端口的设备就可以是 `4-0.1:1.0`；2 号端口的设备就可以是 `4-0.2:1.0`；3 号端口就可以是 `4-0.3:1.0`。总的来说，就是端口号一级一级往下加。

339 行，`state`，设备的状态，这是一个枚举类型。

```
557 enum usb_device_state {
558     /* NOTATTACHED isn't in the USB spec, and this state acts
559      * the same as ATTACHED ... but it's clearer this way.
560      */
561     USB_STATE_NOTATTACHED = 0,
562
563     /* chapter 9 and authentication (wireless) device states */
564     USB_STATE_ATTACHED,
565     USB_STATE_POWERED,                /* wired */
566     USB_STATE_UNAUTHENTICATED,        /* auth */
567     USB_STATE_RECONNECTING,           /* auth */
568     USB_STATE_DEFAULT,                /* limited function */
569     USB_STATE_ADDRESS,
570     USB_STATE_CONFIGURED,             /* most functions */
571
572     USB_STATE_SUSPENDED
573
574     /* NOTE: there are actually four different SUSPENDED
575      * states, returning to POWERED, DEFAULT, ADDRESS, or
576      * CONFIGURED respectively when SOF tokens flow again.
577      */
578 };
```

上面定义了 9 种状态，spec 里只定义了 6 种状态，Attached, Powered, Default, Address, Configured, Suspended，对应于 spec 中的 Table 9.1。

Attached 表示设备已经连接到 USB 接口上了，是 Hub 检测到设备时的初始状态。那么这里所谓的 USB\_STATE\_NOTATTACHED 就是表示设备并没有 Attached。

Powered 是加电状态。USB 设备的电源可以来自外部电源，协议中叫做 self-powered，也可以来自 Hub，称为 bus-powered。尽管 self-powered 的 USB 设备可能在连接上 USB 接口以前已经上电，但它们直到连上 USB 接口后才能被看做是“Powered”的。

Default 默认状态，在 Powered 之后，设备必须在收到一个复位（reset）信号并成功复位后，才能使用默认地址回应主机发过来的设备和配置描述符的请求。

Address 状态表示主机分配了一个唯一的地址给设备，此时设备可以使用默认管道响应主机的请求。

Configured 状态表示设备已经被主机配置过了，也就是协议中说的处理一个带有非 0 值的 SetConfiguration() 请求，此时主机可以使用设备提供的所有功能。

Suspended 挂起状态，为了省电，设备在指定的时间内（大约 3 ms 吧）如果没有发生总线传输，就要进入挂起状态。此时，USB 设备要自己维护包括地址、配置在内的信息。

USB 设备从生到死都要按照这么几个状态，遵循这么一个过程。

340 行，speed，设备的速度，这也是一个枚举变量。

```
550 enum usb_device_speed {
551     USB_SPEED_UNKNOWN = 0,           /* enumerating */
552     USB_SPEED_LOW, USB_SPEED_FULL,   /* usb 1.1 */
553     USB_SPEED_HIGH,                  /* usb 2.0 */
554     USB_SPEED_VARIABLE,              /* wireless (usb 2.5) */
555 };
```

在 USB 3.0 以前，设备有三种速度，低速、全速、高速。USB 1.1 在那时只有低速和全速，后来才出现了高速，就是所谓的 480 MB/s。这里还有一个 USB\_SPEED\_VARIABLE，是无线 USB，号称 USB 2.5。USB\_SPEED\_UNKNOWN 只是表示现阶段还不知道这个设备究竟什么速度。

342 行，tt；343 行，tport。tt 全称为 transaction translator。你可以把它想成一块特殊的电路，是 Hub 里面的电路，确切地说是高速 Hub 中的电路，我们知道 USB 设备有三种速度，分别是 Low Speed, Full Speed, High Speed。即所谓的低速、全速及高速，以前主机控制器，只有两种接口的，OHCI/UHCI，这都是在 USB spec 1.0 时，后来 2.0 推出了 EHCI，高速设备应运而生。

Hub 也有高速的 Hub 和低速的 Hub，但是这里就产生一个兼容性问题了，高速的 Hub 是否能够支持低速/全速的设备呢？一般来说是不支持的，于是有了有一个叫做 TT 的电路，它就负责高速和低速/全速的数据转换。于是，如果一个高速设备中有这么一个 TT，那么就可以连接低速/全速设备，要不然，低速/全速设备就没法用，只能连接到 OHCI/UHCI 那边出来的 Hub 口里。

345 行，`toggle[2]`，这个数组只有两个元素，分别对应 IN 端点和 OUT 端点，每一个端点占一位。似乎这么说仍是在雾中看花，黑格尔告诉我们：“存在就是有价值的。”那么这个数组存在的价值是什么？

前面说，你要想和你的 USB 通信，创建一个 `urb`，为它赋值，交给 USB Core 就可以了。这个 `urb` 是站在我们的角度，实际上在 USB cable 里流淌的根本就不是那么回事儿，我们提交的是 `urb`，USB cable 里流淌的是一个一个的数据包（`packet`）。

所有的 `packets` 都从一个 SYNC 同步字段开始，SYNC 是一个 8 位长的二进制串，只是用来同步用的，它的最后两位标志了 SYNC 的结束和 PID（Packet Identifier）的开始。

PID 也是一个 8 位的二进制串，前四位用来区分不同的 `packet` 类型，后面四位只是前四位的反码，是校验用的。正如 spec 的 Table 8-1 里描述的那样，共有四大类的 `packet`，分别是 Token、Data、Handshake 和 Special，每个大类又包含了几个子类。

主机和设备通过 PID 来判断送过来的 `packet` 是不是自己所需要的。

PID 之后紧跟着的是地址字段，每个 `packet` 都需要知道自己要往哪里去，它们是一个一个目的明确的精灵，行走在 USB cable 里。这个地址实际上包括两部分，7 位表示了总线上连接的设备或接口的地址，4 位表示端点的地址，这就是为什么前面说每条 USB 总线最多只能有 128 个设备，即使是高速设备，除了 0 号端点也最多只能有 15 个 in 端点和 15 个 out 端点。

地址字段再往后是 11 位的帧号（`frame number`），值达到 7FFH 时归零。这个帧号并不是每一个 `packet` 都会有，它只在每帧或微帧（Microframe）开始的 SOF Token 包中发送。帧是对于低速和全速模式来说的，一帧就是 1 ms，对于高速模式的称呼是微帧，一个微帧为 125 ms，每帧或微帧当然不会只能传一个 `packet`。

帧号再往后就是千呼万唤始出来的 Data 字段了，它可以有 0 到 1024 个字节不等。最后还有 CRC 校验字段来做扫尾工作。

我们开始学习 `packet`，但这里只看一看 Data 类型的 `packet`。有四种类型的 Data 包，DATA0，DATA1，DATA2 和 MDATA。存在就是有价值的，这里数据包分成 4 种自然有其道理，其中 DATA0 和 DATA1 就可以用来实现 data toggle 同步，看到 toggle，好像有点接近不久之前留下的疑问了。

对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为 DATA0 的，然

后为了传输的正确性，就传一次 DATA0，再传一次 DATA1，一旦哪次打破了这种规律，主机就可以认为传输出错了。对于等时传输来说，data toggle 并不被支持。USB 就是在使用这种简单的哲学来判断对与错。

我们的 struct usb\_device 中的数组 unsigned int toggle[2]就是为了支持这种简单的哲学而生的，它里面的每一位元素表示的就是每个端点当前发送或接收的数据包是 DATA0 还是 DATA1。

348 行的 parent，struct usb\_device 结构体的 parent 自然也是一个 struct usb\_device 指针。对于 Root Hub，前面说过，它是和 Host Controller 是绑定在一起的。它的 parent 指针在 Host Controller 的驱动程序中就已经赋了值，这个值就是 NULL。换句话说，对于 Root Hub，它不需要再有父指针了，这个父指针就是给从 Root Hub 连出来的节点用的。USB 设备是从 Root Hub 开始，一个一个往外面连。比如 Root Hub 有 4 个接口，每个接口连一个 USB 设备，比如其中有一个是 Hub，那么这个 Hub 有可以继续有多个接口，于是一级一级地往下连，最终连成了一棵树。

349 行，bus，设备所在的那条总线。

350 行，ep0，端点 0 的特殊地位决定了它必将受到特殊的待遇，在 struct usb\_device 对象产生时它就要初始化。

352 行，dev，嵌入到 struct usb\_device 结构中的 struct device 结构。

354 行，desc，设备描述符，四大描述符的第三个姗姗而来。它在 include/linux/usb/ch9.h 中定义。

```
203 /* USB_DT_DEVICE: Device descriptor */
204 struct usb_device_descriptor {
205     __u8  bLength;
206     __u8  bDescriptorType;
207
208     __le16 bcdUSB;
209     __u8  bDeviceClass;
210     __u8  bDeviceSubClass;
211     __u8  bDeviceProtocol;
212     __u8  bMaxPacketSize0;
213     __le16 idVendor;
214     __le16 idProduct;
215     __le16 bcdDevice;
216     __u8  iManufacturer;
217     __u8  iProduct;
218     __u8  iSerialNumber;
219     __u8  bNumConfigurations;
220 } __attribute__((packed));
221
222 #define USB_DT_DEVICE_SIZE 18
```

205 行，bLength，描述符的长度，可以自己数一数，或者看紧接着的定义 USB\_DT\_DEVICE\_SIZE。

206 行, `bDescriptorType`, 这里对于设备描述符应该是 `USB_DT_DEVICE`, `0x01`。

208 行, `bcdUSB`, USB spec 的版本号, 一个设备如果能够进行高速传输, 那么它的设备描述符里的 `bcdUSB` 这一项就应该为 `0200H`。

209 行, `bDeviceClass`; 210 行, `bDeviceSubClass`; 211 行, `bDeviceProtocol`, 和接口描述符的意义差不多。

212 行, `bMaxPacketSize0`, 端点 0 一次可以处理的最大字节数, 端点 0 的属性却放到设备描述符里去了, 更加彰显了它突出的地位。

前面说端点时说了端点 0 并没有一个专门的端点描述符, 因为不需要, 基本上它所有的特性都在 spec 里规定好了的。然而, 别忘了这里说的是“基本上”, 有一个特性则是不一样的, 这叫做 `maximum packet size`, 每个端点都有个特性, 即告诉你该端点能够发送或者接收的包的最大值。对于通常的端点来说, 这个值被保存在该端点描述符中的 `wMaxPacketSize` 这个 field, 而对于端点 0 就不一样了, 由于它自己没有一个描述符, 而每个设备又都有这么一个端点, 所以这个信息被保存在了设备描述符里, 所以我们在设备描述符里可以看到, `bMaxPacketSize0`。而且 spec 还规定了, 这个值只能是 8, 16, 32 或者 64 这四者之一, 如果一个设备工作在高速模式, 这个值还只能是 64, 如果是工作在低速模式, 则只能是 8, 取别的值都不行。

213 行, `idVendor`; 214 行, `idProduct`, 分别是厂商和产品的 ID 号。

215 行, `bcdDevice`, 设备的版本号。

216 行, `iManufacturer`; 217 行, `iProduct`; 218 行, `iSerialNumber`, 分别是厂商, 产品和序列号对应的字符串描述符的索引值。

219 行, `bNumConfigurations`: 设备当前速度模式下支持的配置数量。有的设备可以在多个速度模式下操作, 这里包括的只是当前速度模式下的配置数目, 不是总的配置数目。

这就是设备描述符, 它和 spec 中的 Table 9-8 是一一对应的。回到 `struct usb_device` 的 355 行, `config`; 357 行, `actconfig`, 分别表示设备拥有的所有配置和当前激活的, 也就是正在使用的配置。USB 设备的配置用 `struct usb_host_config` 结构来表示, 下节再说。

358 行, `ep_in[16]`; 359 行, `ep_out[16]`, 除了端点 0, 一个设备即使在高速模式下也最多只能再有 15 个 IN 端点和 15 个 OUT 端点, 端点 0 太特殊了, 对应的管道是 `message` 管道, 又能进又能出, 所以这里的 `ep_in` 和 `ep_out` 数组都有 16 个值。

361 行, `rawdescriptors`, 这是一个字符指针数组, 数组里的每一项都指向一个使用 `GET_DESCRIPTOR` 请求去获得配置描述符时所得到的结果。考虑一下, 为什么我只说得到的结果, 而不直接说得到的配置描述符? 不是请求的就是配置描述符吗? 这是因为当你使用 `GET_DESCRIPTOR` 去请求配置描述符时, 设备返回给你的不仅仅只有配置描述符, 它把该配

置所包括的所有接口的接口描述符，还有接口里端点的端点描述符一股脑儿地都塞给你了。第一个接口的接口描述符紧跟着这个配置描述符，然后是这个接口下面端点的端点描述符，如果有还有其他接口，它们的接口描述符和端点描述符也跟在后面，这里面专门为一类设备定义的描述符和厂商定义的描述符跟在它们对应的标准描述符后面。

这里提到了 GET\_DESCRIPTOR 请求，就顺便简单提一下 USB 的设备请求(device request)。协议中说了，所有的设备通过默认的控制管道来响应主机的请求，既然使用的是控制管道，那当然就是控制传输了，这些请求的底层 packet 属于 Setup 类型。协议中同时也定义了一些标准的设备请求，并规定所有的设备必须响应它们，即使它们还处于 Default 或 Address 状态。在这些标准的设备请求里，GET\_DESCRIPTOR 就赫然在列。

363 行，bus\_mA，这个值是在主机控制器的驱动程序中设置的，通常来讲，计算机的 USB 端口可以提供 500mA 的电流。

364 行，portnum，不管是 Root Hub 还是一般的 Hub，你的 USB 设备总归要插在一个 Hub 的端口上才能用，portnum 就是那个端口号。当然，对于 Root Hub 这个 USB 设备来说，它本身没有 portnum 这个概念，因为它不插在别的 Hub 的任何一个口上。所以对于 Root Hub 来说，它的 portnum 在主机控制器的驱动程序里给设置成了 0。

365 行，level，层次，也可以说是级别，表示 USB 设备树的级连关系。Root Hub 的 level 当然就是 0，其下面一层就是 level 1，再下面一层就是 level 2，依此类推。

366 行，discon\_suspended，Disconnected while suspended。

368 行，have\_langid; 369 行，string\_langid，USB 设备中的字符串描述符使用的是 UNICODE 编码，可以支持多种语言，string\_langid 就是用来指定使用哪种语言的，have\_langid 用来判断 string\_langid 是否有效。

372 行，product; 373 行，manufacturer; 374 行，serial，分别用来保存产品、厂商和序列号对应的字符串描述符信息。

376 行到 382 行，与 usbfs 相关的代码。

391 行，maxchild，Hub 的端口数，注意可不包括上行端口。

392 行，children[USB\_MAXCHILDREN]，USB\_MAXCHILDREN 是 include/linux/usb.h 中定义的一个宏，值为 31。

```
324 #define USB_MAXCHILDREN (31)
```

其实 Hub 可以一共接 255 个端口，不过实际上遇到的 Hub 最多的也就是支持 10 个端口的，所以 31 端口基本上够用了。

394 行, `pm_usage_cnt`, `struct usb_interface` 结构中也有。

396 行, `quirks`, 简单说就是 “毛病”。

397 行, 看到 `#ifdef CONFIG_PM` 这个标志, 我们就知道从这里直到最后的那个 `#endif` 都是关于电源管理的代码。

## 18. 配置

还是接着看 USB 设备的配置吧, 在 `include/linux/usb.h` 文件中定义:

```
244 struct usb_host_config {
245     struct usb_config_descriptor    desc;
246
247     char *string;                    /* iConfiguration string, if present */
248     /* the interfaces associated with this configuration,
249      * stored in no particular order */
250     struct usb_interface *interface[USB_MAXINTERFACES];
251
252     /* Interface information available even when this is not the
253      * active configuration */
254     struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
255
256     unsigned char *extra;            /* Extra descriptors */
257     int extralen;
258 };
```

245 行, `desc`, 四大描述符里最后的一个描述符终于出现了, 同样是在 `include/linux/usb/ch9.h` 中定义。

```
258 struct usb_config_descriptor {
259     __u8    bLength;
260     __u8    bDescriptorType;
261
262     __le16  wTotalLength;
263     __u8    bNumInterfaces;
264     __u8    bConfigurationValue;
265     __u8    iConfiguration;
266     __u8    bmAttributes;
267     __u8    bMaxPower;
268 } __attribute__((packed));
269
270 #define USB_DT_CONFIG_SIZE          9
```

259 行, `bLength`, 描述符的长度, 值为 `USB_DT_CONFIG_SIZE`。

260 行, `bDescriptorType`, 描述符的类型, 值为 `USB_DT_CONFIG`, `0x02`。按照前面接口描述符、端点描述符和设备描述符的习惯来说, 这样说应该是没问题。但是, 这里的值却并不



仅仅可以为 USB\_DT\_CONFIG，还可以为 USB\_DT\_OTHER\_SPEED\_CONFIG，0x07。这里说的 OTHER\_SPEED\_CONFIG 描述符描述的是高速设备操作在低速或全速模式时的配置信息，和配置描述符的结构完全相同，区别只是描述符的类型不同，是只有名字不同的孪生兄弟。

262 行，wTotalLength，是使用 GET\_DESCRIPTOR 请求从设备中获得配置描述符信息时，返回的数据长度，也就是说对包括配置描述符、接口描述符、端点描述符，class-或 vendor-specific 描述符在内的所有描述符进行统计。

263 行，bNumInterfaces，这个配置包含的接口数目。

263 行，bConfigurationValue，对于拥有多个配置的幸运设备来说，可以用这个值为参数，使用 SET\_CONFIGURATION 请求来改变正在被使用的 USB 配置，bConfigurationValue 就指明了将要激活哪个配置。设备虽然可以有多个配置，但同一时间却也只能有一个配置被激活。顺便说一下，SET\_CONFIGURATION 请求也是标准的设备请求之一，专门用来设置设备的配置。

265 行，iConfiguration，描述配置信息的字符串描述符的索引值。

266 行，bmAttributes，这个字段表示了配置的一些特点，比如 bit 6 为 1 表示 self-powered，bit 5 为 1 表示这个配置支持远程唤醒。另外，它的 bit 7 必须为 1，ch9.h 里有几个相关的定义：

```
272 /* from config descriptor bmAttributes */
273 #define USB_CONFIG_ATT_ONE (1 << 7) /* must be set */
274 #define USB_CONFIG_ATT_SELFPOWER (1 << 6) /* self powered */
275 #define USB_CONFIG_ATT_WAKEUP (1 << 5) /* can wakeup */
276 #define USB_CONFIG_ATT_BATTERY (1 << 4) /* battery powered */
```

267 行，bMaxPower，设备正常运转时，从总线那里分得的最大电流值，以 2mA 为单位。设备可以使用这个字段向 Hub 表明自己需要的电流，但如果设备需求过于多，请求超出了 Hub 所能给予的，Hub 就会直接拒绝。还记得 struct usb\_device 结构中的 bus\_mA 吗？它就表示 Hub 所能够给予的。Alan Stern 告诉我们：

```
(c->desc.bMaxPower * 2) is what the device requests and udev->bus_mA is what the
hub makes available.
```

到此为止，四大标准描述符已经全部登场亮相了，还是回到 struct usb\_host\_config 结构的 247 行，string，这个字符串保存了配置描述符 iConfiguration 字段对应的字符串描述符信息。

250 行，interface[USB\_MAXINTERFACES]，配置所包含的接口。注释里说的很明确，这个数组的顺序未必是按照配置里接口号的顺序，所以你要想得到某个接口号对应的 struct usb\_interface 结构对象，就必须使用 drivers/usb/usb.c 中定义的 usb\_ifnum\_to\_if 函数。

```
84 struct usb_interface *usb_ifnum_to_if(const struct usb_device *dev,
85                                     unsigned ifnum)
86 {
87     struct usb_host_config *config = dev->actconfig;
88     int i;
```

```

89
90     if (!config)
91         return NULL;
92     for (i = 0; i < config->desc.bNumInterfaces; i++)
93         if (config->interface[i]->altsetting[0]
94             .desc.bInterfaceNumber == ifnum)
95             return config->interface[i];
96
97     return NULL;
98 }

```

这个函数的道理很简单，就是拿你指定的接口号，和当前配置的每一个接口可选设置 0 里接口描述符的 `bInterfaceNumber` 字段做比较，如果相等，那个接口就是你要寻找的，如果都不相等，不能满足你的要求，虽然它已经尽力了。

如果你看了协议，可能会在 spec 的 9.6.5 节里看到，请求配置描述符时，配置里的所有接口描述符是按照顺序一个一个返回的。那为什么这里又明确说明，不要期待它会是接口号的顺序？其实以前这里并不是这么说的，这个数组是按照 `0..desc.bNumInterfaces` 的顺序，但同时又需要通过 `usb_ifnum_to_if` 函数来获得指定接口号的接口对象，Alan Stern 质疑了这种有些矛盾的说法，于是 David Brownell 就把它改成现在这个样子了，为什么改？因为协议归协议，厂商归厂商，有些厂商就是不遵守协议，它非要先返回接口 1 再返回接口 0，所以就不得不增加 `usb_ifnum_to_if` 函数。

`USB_MAXINTERFACES` 是 `drivers/usb/usb.h` 中定义的一个宏，值为 32，不要说不够用，谁见过有很多接口的设备？

```

/* this maximum is arbitrary */
#define USB_MAXINTERFACES    32

```

254 行，`intf_cache[USB_MAXINTERFACES]`，`cache` 是缓存。这是一个 `struct usb_interface_cache` 对象的结构数组；`usb_interface`，USB 接口；`cache`，缓存。所以 `usb_interface_cache` 就是 USB 接口的缓存。缓存些什么？在 `include/linux/usb/usb.h` 里查看其定义。

```

193 struct usb_interface_cache {
194     unsigned num_altsetting;          /* number of alternate settings */
195     struct kref ref;                 /* reference counter */
196
197     /* variable-length array of alternate settings for this interface,
198      * stored in no particular order */
199     struct usb_host_interface altsetting[0];
200 };

```

199 行的 `altsetting[0]` 是一个可变长数组，可以按需分配，你对设备说 `GET_DESCRIPTOR` 时，内核就根据返回的每个接口可选设置的数目分配给 `intf_cache` 数组相应的空间，有多少需要就分配多少空间。

为什么要缓存这些东西？为了在配置被取代之后仍然能够获取它的一些信息，就把日后可

能会需要的一些东西放在了 `intf_cache` 数组的 `struct usb_interface_cache` 对象里。谁会需要？这么说吧，你通过 `sysfs` 这个窗口只能看到设备当前配置的一些信息，即使是这个配置下面的接口，也只能看到接口正在使用的可选设置的信息，可是你希望能够看到更多的信息，怎么办？`usbfs` 里面显示有系统中所有 USB 设备的可选配置和端点信息，它就是利用 `intf_cache` 这个数组里缓存的东西实现的。

256 行，`extra`；257 行，`extralen`，有关额外扩展的描述符，和 `struct usb_host_interface` 里的差不多，只是这里的是针对配置的，如果你使用 `GET_DESCRIPTOR` 请求从设备中获得配置描述符信息，它们会紧跟在标准的配置描述符后面返回给你。

---

## 19. 向左走，向右走

我们回到前面提到的函数 `usb_device_match`：

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545         /* interface drivers never match devices */
546         if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551     } else {
552         struct usb_interface *intf;
553         struct usb_driver *usb_drv;
554         const struct usb_device_id *id;
555
556         /* device drivers never match interfaces */
557         if (is_usb_device_driver(drv))
558             return 0;
559
560         intf = to_usb_interface(dev);
561         usb_drv = to_usb_driver(drv);
562
563         id = usb_match_id(intf, usb_drv->id_table);
564         if (id)
565             return 1;
566
567         id = usb_match_dynamic_id(intf, usb_drv);
568         if (id)
569             return 1;
570     }
571 }
572
573 return 0;
```

```
574 }
```

在 USB 的世界里，对于设备和驱动来说只是 `usb_device_match` 函数的两端。`usb_device_match` 函数为它们指明向左走还是向右走。

543 行，第一次遇到这个函数时，我说了这里有两条路，一条给 USB 设备走，一条给 USB 接口走。先来查看设备走的这条路，上面只有两个函数。

```
85 static inline int is_usb_device(const struct device *dev)
86 {
87     return dev->type == &usb_device_type;
88 }
```

`drivers/usb/core/usb.h` 中定义的这个函数就是要告诉你，只有 `usb_device` 才能从这里走。但关键问题不是它让不让你进去，而是它怎么知道你是不是 `usb_device`。

关键就在于这个 `dev->type`，设备的类型，看它是不是等于定义的 `usb_device_type`，也就是说 USB 设备类型，相等的话，那可以通过；不相等，那就此路不是为你开的，你找别的路吧。`usb_device_type` 在 `drivers/usb/core/usb.c` 中定义：

```
195 struct device_type usb_device_type = {
196     .name = "usb_device",
197     .release = usb_release_dev,
198 };
```

它和前面看到的 `usb_bus_type` 差不多，一个表示总线的类型，一个表示设备的类型，总线有总线的类型，设备有设备的类型。

假设现在过来一个设备，经过判断，它要走的是设备这条路，可问题是，这个设备的 `type` 字段什么时候被初始化成 `usb_device_type` 了。这倒是一个问题，不过先不说明，继续向前走，带着疑问上路。

546 行，又见到一个 `if`，它就在 `is_usb_device` 函数后面在 `drivers/usb/core/usb.h` 文件中定义：

```
92 static inline int is_usb_device_driver(struct device_driver *drv)
93 {
94     return container_of(drv, struct usbdrv_wrap, driver)->
95         for_devices;
96 }
```

这个函数用于判断是不是 `usb device driver`，那什么是 `usb device driver`？前面不是一直都是说一个 USB 接口对应一个 USB 驱动吗？可是我们不能只钻到接口的那个接口里边儿，我们应该眼光放得更加开阔一些，要知道接口在 USB 的世界里并不是老大，它上边儿还有配置，还有设备，都比它大。

每个接口对应了一个独立的功能，是需要专门的驱动来和它交流，但是接口毕竟整体是作为一个 USB 设备而存在的，设备还可以有不同的配置，我们还可以为设备指定特定的配置，那

谁来做这个事情？`struct usb_device _driver`，即 USB 设备驱动，它和 USB 的接口驱动 `struct usb_driver` 都定义在 `include/linux/usb.h` 文件中。

```
833 struct usb_driver {
834     const char *name;
835
836     int (*probe) (struct usb_interface *intf,
837                  const struct usb_device_id *id);
838
839     void (*disconnect) (struct usb_interface *intf);
840
841     int (*ioctl) (struct usb_interface *intf, unsigned int code,
842                  void *buf);
843
844     int (*suspend) (struct usb_interface *intf, pm_message_t message);
845     int (*resume) (struct usb_interface *intf);
846
847     void (*pre_reset) (struct usb_interface *intf);
848     void (*post_reset) (struct usb_interface *intf);
849
850     const struct usb_device_id *id_table;
851
852     struct usb_dynids dynids;
853     struct usbdrv_wrap drvwrap;
854     unsigned int no_dynamic_id:1;
855     unsigned int supports_autosuspend:1;
856 };
```

一般来说，我们平时所谓的编写 USB 驱动指的也就是写 USB 接口的驱动，需要以一个 `struct usb_driver` 结构的对象为中心，以设备的接口提供的功能为基础，开展 USB 驱动的建设。

834 行，`name`，驱动程序的名字，对应了在 `/sys/bus/usb/drivers/` 下面的子目录名称。和我们每个人一样，它只是区别彼此的一个代号，不同的是我们可以有很多人同名，但这里的名字在所有的 USB 驱动中必须是唯一的。

836 行，`probe`，用来看一看这个 USB 驱动是否愿意接受某个接口的函数。一个驱动往往可以支持多个接口。

839 行，`disconnect`，当接口失去联系，或使用 `rmmod` 卸载驱动将它和接口强行分开时这个函数就会被调用。

841 行，`ioctl`，当你的驱动需要通过 `usbfs` 和用户空间交流的话，就可以使用 `ioctl`。

844 行，`suspend`；845 行，`resume`，分别在设备被挂起和唤醒时使用。

847 行，`pre_reset`；848 行，`post_reset`，分别在设备将要复位（`reset`）和已经复位后使用。

850 行，`id_table`，驱动支持的所有设备的列表，驱动就靠这张列表来识别是不是支持哪个设备接口的，如果不属于这张列表，那就不支持。

852 行, `dynids`, 支持动态 id。什么是动态 id? 每个驱动诞生时它的 id 在 `id_table` 里就已经确定了, 可是 Greg 显然在一年多前加入了动态 id 的机制。即使驱动已经加载了, 也可以添加新的 id 给它, 只要新 id 代表的设备存在, 就会和它绑定起来。

怎么添加新的 id? 到驱动所在的地方看一看, 也就是 `/sys/bus/usb/drivers` 目录里面, 那里列出的每个目录就代表了一个 USB 驱动, 随便选一个进去, 能够看到一个 `new_id` 文件, 使用 `echo` 将厂商和产品 id 写进去就可以了。查看 Greg 举的一个例子:

```
echo 0557 2008 > /sys/bus/usb/drivers/foo_driver/new_id
```

可以将十六进制值 0557 和 2008 写到 `foo_driver` 驱动的设备 id 表里。

853 行, `drvwrap`, 这个字段是 `struct usbdrv_wrap` 结构, 也在 `include/linux/usb.h` 中定义。

```
779 struct usbdrv_wrap {
780     struct device_driver driver;
781     int for_devices;
782 };
```

近距离观察一下这个结构, 它里面内容比较贫乏, 只有一个 `struct device_driver` 结构的对象和一个 `for_devices` 的整型字段。回想一下 Linux 的设备模型, 我们就会产生这样的疑问, 这里的 `struct device_driver` 对象不是应该嵌入到 `struct usb_driver` 结构中吗, 怎么这里又包装了一层?

再包装这么一层当然不是为了美观, 这主要还是因为本来驱动在 USB 的世界里不得已分成了设备驱动和接口驱动两种, 为了区分这两种驱动, 就在中间加了这么一层, 添了一个 `for_devices` 标志来判断是哪种驱动。

不知你有没有发现, 之前见识过的结构中, 很多不是 1 就是 0 的标志使用的是位字段, 特别是几个这样的标志放一起时, 而这里的 `for_devices` 虽然只能有两个值, 但却没有使用位字段, 为什么? 因为没必要, 那些使用位字段的是几个在一块儿, 可以节省点儿存储空间, 而这里只有这么一个, 就是使用位字段也节省不了, 就不用多此一举了。

其实就这么说为了加一个判断标志就硬生生地加这么一层, 不过, 这个 `drvwrap` 以后肯定还会遇到它, 这里先简单介绍一下。

854 行, `no_dynamic_id`, 可以用来禁止动态 id 的。

855 行, `supports_autosuspend`, 对 `autosuspend` 的支持, 如果设置为 0 的话, 就不再允许绑定到这个驱动的接口 `autosuspend`。

`struct usb_driver` 结构就暂时了解到这里, 再来看一看所谓的 USB 设备驱动与接口驱动的区别。

```
878 struct usb_device_driver {
```

```

879     const char *name;
880
881     int (*probe) (struct usb_device *udev);
882     void (*disconnect) (struct usb_device *udev);
883
884     int (*suspend) (struct usb_device *udev, pm_message_t message);
885     int (*resume) (struct usb_device *udev);
886     struct usbdrv_wrap drvwrap;
887     unsigned int supports_autosuspend:1;
888 };

```

这个 USB 设备驱动比前面的接口驱动要简洁多了，除了少了很多参数外，剩下的将参数中的 `struct usb_interface` 换成 `struct usb_device` 后就几乎一模一样了。

提醒一下，这里说的是几乎，而不是完全，这是因为 `probe`，它的参数中与接口驱动里的 `probe` 相比少了设备的列表，也就是说它不用再去根据列表来判断是不是愿意接受一个 USB 设备。那么这意味着什么？是它来者不拒，接受所有的 USB 设备？还是拒绝所有的 USB 设备？当然只会是前者，不然这个 USB 设备驱动就完全毫无疑问了，而且我们在内核中来找去，也只能找到它在 `drivers/usb/core/generic.c` 文件中定义了 `usb_generic_driver` 这个对象：

```

210 struct usb_device_driver usb_generic_driver = {
211     .name = "usb",
212     .probe = generic_probe,
213     .disconnect = generic_disconnect,
214 #ifdef CONFIG_PM
215     .suspend = generic_suspend,
216     .resume = generic_resume,
217 #endif
218     .supports_autosuspend = 1,
219 };

```

这个对象在 `usb_init` 的 895 行就已经注册给 USB 子系统了。

不管怎么说，总算是把 USB 接口的驱动和设备的驱动给讲了一下，还是回到这节开头的 `usb_device_match` 函数，目前为止，设备走的这条路已经比较清晰了，就是如果设备过来了，走到了设备这条路，然后要判断一下驱动是不是设备驱动，是不是针对整个设备的，如果不是的话，对不起，虽然这条路走对了，可是沿这条路，设备找不到对应的驱动，匹配不成功，就直接返回了。那如果驱动也确实是设备驱动呢？代码直接返回 1，表示匹配成功了。

本来这么说应该是可以了，可是我还是忍不住想告诉你，在之前的内核版本里，是没有很明确的 `struct usb_device_driver` 这样一个表示 USB 设备驱动的结构，而是直接定义了 `struct device_driver` 结构类型的一个对象 `usb_generic_driver` 来处理与整个设备相关的事情，相对应的，`usb_device_match` 这个匹配函数也只有简单的一条路，在接口和对应的驱动之间做匹配。但是在 2006 年，内核中多了 `struct usb_device_driver` 结构，`usb_device_match` 这里也多了一条给设备走的路。

是时候也有必要对 USB 设备在 USB 世界里的整个人生旅程做一个介绍了。

## 20. 设备的生命线（一）

设备也有它自己的生命线，当你把它插到 Hub 上开始，到你把它从 Hub 上拔下来结束。

当然你将 USB 设备连接在 Hub 的某个端口上，Hub 检测到有设备连接了进来，它会为设备分配一个 `struct usb_device` 结构的对象并初始化，调用设备模型提供的接口将设备添加到 USB 总线的设备列表里，然后 USB 总线会遍历驱动列表里的每个驱动，调用自己的 `match` 函数看它们和你的设备或接口是否匹配。又走到 `match` 函数了。

Hub 检测到自己的某个端口有设备连接了进来后，它会调用 `core` 里的 `usb_alloc_dev` 函数为 `struct usb_device` 结构的对象申请内存，这个函数在 `drivers/usb/core/usb.c` 文件中定义。

```

238 struct usb_device *
239 usb_alloc_dev(struct usb_device *parent, struct usb_bus *bus, unsigned
port1)
240 {
241     struct usb_device *dev;
242
243     dev = kzalloc(sizeof(*dev), GFP_KERNEL);
244     if (!dev)
245         return NULL;
246
247     if (!usb_get_hcd(bus_to_hcd(bus))) {
248         kfree(dev);
249         return NULL;
250     }
251
252     device_initialize(&dev->dev);
253     dev->dev.bus = &usb_bus_type;
254     dev->dev.type = &usb_device_type;
255     dev->dev.dma_mask = bus->controller->dma_mask;
256     dev->state = USB_STATE_ATTACHED;
257
258     INIT_LIST_HEAD(&dev->ep0.urb_list);
259     dev->ep0.desc.bLength = USB_DT_ENDPOINT_SIZE;
260     dev->ep0.desc.bDescriptorType = USB_DT_ENDPOINT;
261     /* ep0 maxpacket comes later, from device descriptor */
262     dev->ep_in[0] = dev->ep_out[0] = &dev->ep0;
263
264     /* Save readable and stable topology id, distinguishing devices
265     * by location for diagnostics, tools, driver model, etc. The
266     * string is a path along hub ports, from the root. Each device's
267     * dev->devpath will be stable until USB is re-cabled, and hubs
268     * are often labeled with these port numbers. The bus_id isn't
269     * as stable: bus->busnum changes easily from modprobe order,
270     * cardbus or pci hotplugging, and so on.
271     */
272     if (unlikely(!parent)) {
273         dev->devpath[0] = '';
274
275         dev->dev.parent = bus->controller;
276         sprintf(&dev->dev.bus_id[0], "usb%d", bus->busnum);
277     } else {

```



```

278      /* match any labeling on the hubs; it's one-based */
279      if (parent->devpath[0] == '/')
280          snprintf(dev->devpath, sizeof dev->devpath,
281                  "%d", port1);
282      else
283          snprintf(dev->devpath, sizeof dev->devpath,
284                  "%s.%d", parent->devpath, port1);
285
286      dev->dev.parent = &parent->dev;
287      sprintf(&dev->dev.bus_id[0], "%d-%s",
288              bus->busnum, dev->devpath);
289
290      /* hub driver sets up TT records */
291  }
292
293  dev->portnum = port1;
294  dev->bus = bus;
295  dev->parent = parent;
296  INIT_LIST_HEAD(&dev->filelist);
297
298  #ifdef CONFIG_PM
299      mutex_init(&dev->pm_mutex);
300      INIT_DELAYED_WORK(&dev->autosuspend, usb_autosuspend_work);
301      dev->autosuspend_delay = usb_autosuspend_delay * HZ;
302  #endif
303      return dev;
304  }

```

`usb_alloc_dev` 函数就相当于 USB 设备的构造函数，在参数中，`parent` 是设备连接的 Hub，`bus` 是设备连接的总线，`port1` 就是设备连接在 Hub 上的端口。

243 行，为一个 `struct usb_device` 结构的对象申请内存并初始化为 0。直到在看到这一行前，我还仍在使用 `kmalloc` 加 `memset` 这对最佳拍档来申请内存和初始化，但是在看到 `kzalloc` 之后，我知道了 `kzalloc` 直接取代了 `kmalloc/memset`，一个函数起到了两个函数的作用。

然后判断内存申请是否成功，如果不成功就不用往下走了。那么通过这几行，我们应该记住，凡是你想用 `kmalloc/memset` 组合申请内存时，就使用 `kzalloc` 代替吧；凡是申请内存的，不要忘了判断是否申请成功了。

247 行，这里的两个函数是 `hcd`，是主机控制器驱动里的。要知道 USB 的世界里一个主机控制器对应着一条 USB 总线，主机控制器驱动用 `struct usb_hcd` 结构表示，一条总线用 `struct usb_bus` 结构表示，函数 `bus_to_hcd` 是为了获得总线对应的主机控制器驱动，也就是 `struct usb_hcd` 结构对象，函数 `usb_get_hcd` 只是将得到的 `usb_hcd` 结构对象的引用计数加 1。为什么？因为总线上多了一个设备，当然得为它增加引用计数。如果这两个函数没有很好地完成自己的任务，那整个 `usb_alloc_dev` 函数也就没有必要继续执行下去了，将之前为 `struct usb_device` 结构对象申请的内存释放掉就可以了。

252 行，`device_initialize` 是设备模型中的函数，第一个 `dev` 是 `struct usb_device` 结构体指针，而第二个 `dev` 是 `struct device` 结构体，这是设备模型中一个最基本的结构体，使用它必然

要先初始化。`device_initialize` 函数的目的就是将在 `struct usb_device` 结构中嵌入的 `struct device` 结构体初始化掉，方便以后调用。

253 行，将设备所在的总线类型设置为 `usb_bus_type`。`usb_bus_type` 在前面见过了，USB 子系统的初始化函数 `usb_init` 里就把它给注册掉了。

254 行，将设备的设备类型初始化为 `usb_device_type`，这是在上节第二次遇到 `usb_device_match` 函数，走设备那条路时，使用 `is_usb_device` 判断是不是 USB 设备时留下的疑问，就是在这儿把设备的类型给初始化成 `usb_device_type` 了。

255 行，就是与 DMA 传输相关的了，设备能不能进行 DMA 传输，得看主机控制器的脸色，主机控制器不支持设备也没法使用。所以这里 `dma_mask` 被设置为主机控制器的 `dma_mask`。

256 行，将 USB 设备的状态设置为 `ATTACHED`，表示设备已经连接到 USB 接口上了，是 Hub 检测到设备时的初始状态。

258 行，端点 0 实在是太特殊了，`struct usb_device` 里直接就有一个 `ep0` 成员，这行就将 `ep0` 的 `urb_list` 给初始化掉了。

259 行，260 行，分别初始化了端点 0 的描述符长度和描述符类型。

260 行，使 `struct usb_device` 结构中的 `ep_in` 和 `ep_out` 指针数组的第一个成员指向 `ep0`，`ep_in[0]` 和 `ep_out[0]` 本来表示的就是端点 0。

272 行，这里平白无故地多出了一个 `unlikely`，不知道什么意思？先查看它们在 `include/linux/compiler.h` 的定义。

```
60 #define likely(x)      __builtin_expect(!!(x), 1)
61 #define unlikely(x)    __builtin_expect(!!(x), 0)
```

除了函数 `unlikely`，还有一个函数 `likely`。定义里 `__builtin_expect` 是 GCC 里内建的一个函数，具体是做什么用的可以看一看 GCC 的手册。

```
long __builtin_expect (long exp, long c)
You may use __builtin_expect to provide the compiler with branch prediction
information. In general, you should prefer to use actual profile feedback for this
('-fprofile-arcs'), as programmers are notoriously bad at predicting how their progra
ms actually perform. However, there are applications in which this data is hard to
collect.
The return value is the value of exp, which should be an integral expression.
The value of c must be a compile-time constant. The semantics of the built-in are
that it is expected that exp == c. For example:
if (__builtin_expect (x, 0))
    foo ();
would indicate that we do not expect to call foo, since we expect x to be zero.
Since
you are limited to integral expressions for exp, you should use constructions
such
```

```
as
if (__builtin_expect (ptr != NULL, 1))
    error ();
when testing pointer or floating-point values.
```

其大致意思就是由于大部分写代码的人在分支预测方面做的比较糟糕，所以 GCC 提供了这个内建的函数来帮助处理分支预测，优化程序，它的第一个参数 `exp` 为一个整型的表达式，返回值也是这个 `exp`，它的第二个参数 `c` 的值必须是一个编译器的常量，那这个内建函数的意思就是 `exp` 的预期值为 `c`，编译器可以根据这个信息适当地重排条件语句块的顺序，将符合这个条件的分支放在合适的地方。

具体到 `unlikely(x)` 就是告诉编译器条件 `x` 发生的可能性不大，那么这个条件块里语句的目标码可能就会被放在一个比较远的位置，以保证经常执行的目标码更紧凑。而 `likely` 则相反。

这个函数使用时还是很简单的，直接使用 `if` 语句，只是如果你觉得 `if` 条件为 1 的可能性非常大时，可以在条件表达式外面包装一个 `likely()`，如果可能性非常小，则用 `unlikely()` 包装。那么这里 272 行的意思就很明显了，就是说写内核的人觉得你的 USB 设备直接连接到 root hub 上的可能性比较小，因为 `parent` 指的就是你的设备连接的那个 Hub。

272 行到 291 行整个的代码就是首先判断你的设备是不是直接连到 Root Hub 上的，如果是，将 `dev->devpath[0]` 赋值为 0，以示特殊，然后父设备设为 `controller`，同时把 `dev->bus_id[]` 设置为如 `usb1/usb2/usb3/usb4` 这样的字符串。如果你的设备不是直接连到 Root Hub 上的，分两种情况：如果你的设备连接的 Hub 是直接连到 Root Hub 上的，则 `dev->devpath` 就等于端口号，否则 `dev->devpath` 就等于在父 Hub 的 `devpath` 基础上加一个 “/” 再加一个端口号，最后把 `bus_id[]` 设置成 1-/2-/3-/4-这样的字符串后面连接上 `devpath`。

296 行，初始化一个队列。

298 行到 302 行，用于电源管理。

## 21. 设备的生命线（二）

现在设备的 `struct usb_device` 结构体已经准备好了，只是还不怎么饱满，Hub 接下来就会给它做整容手术，往里边儿塞点什么，充实一些内容，比如：将设备的状态设置为 `Powered`，也就是加电状态；因为此时还不知道设备支持的速度，于是将设备的 `speed` 成员暂时先设置为 `USB_SPEED_UNKNOWN`；设备的级别 `level` 当然会被设置为 Hub 的 `level` 加上 1；还有为设备能够从 Hub 那里获得的电流赋值；为了保证通信畅通，Hub 还会为设备在总线上选择一个独一无二的地址。

表 1.20.1

Devnum	Taken
devpath[16]	Taken
State	USB_STATE_POWERED
Speed	USB_SPEED_UNKNOWN
Parent	设备连接的那个 hub
Bus	设备连接的那条总线
ep0	ep0.urb_list, 描述符长度/类型
Dev	dev.bus, dev.type, dev.dma_mask,, dev.parent, dev.bus_id
ep_in[16]	ep_in[0]
ep_out[16]	ep_out[0]
bus_mA	hub->mA_per_port
Portnum	设备连接在 hub 上的那个端口
Level	Hdev->level + 1
Filelist	Taken
pm_mutex	Taken
Autosuspend	Taken
autosuspend_delay	2 * HZ

前面讲过，设备要想从 Powered 状态发展到下一个 Default 状态，必须收到一个复位信号并成功复位。那 Hub 接下来的动作就很明显了，复位设备，复位成功后，设备就会进入 Default 状态。

现在就算设备成功复位了，进入了 Default 状态，同时，Hub 也会获得设备真正的速度。那根据这个速度，起码能够知道端点 0 一次能够处理的最大数据长度，协议中规定，对于高速设备，这个值为 64 字节，对于低速设备，这个值为 8 字节，而对于全速设备可能为 8 字节、16 字节、32 字节、64 字节其中的一个。

设备也该进入 Address 状态了。

设备要想进入 Address 状态很容易，只要 Hub 使用 core 中定义的一个函数 usb\_control\_msg，发送 SET\_ADDRESS 请求给设备，设备就进入 Address 状态了。设备的 address，就是上面的 Devnum。

那现在咱就来说说 usb\_control\_msg 函数，它在 drivers/usb/core/message.c 中定义。

```
120 int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
__u8 requesttype,
121         __u16 value, __u16 index, void *data, __u16 size, int timeout)
122 {
123     struct usb_ctrlrequest *dr = kmalloc(sizeof(struct usb_ctrlrequest),
GFP_NOIO);
124     int ret;
125
126     if (!dr)
127         return -ENOMEM;
128
```

```

129     dr->bRequestType= requesttype;
130     dr->bRequest = request;
131     dr->wValue = cpu_to_le16p(&value);
132     dr->wIndex = cpu_to_le16p(&index);
133     dr->wLength = cpu_to_le16p(&size);
134
135     //dbg("usb_control_ msg");
136
137     ret = usb_internal_control_ msg(dev, pipe, dr, data, size, timeout);
138
139     kfree(dr);
140
141     return ret;
142 }

```

这个函数主要目的是创建一个控制 **urb**，并把它发送给 USB 设备，然后等待它完成。**urb** 是什么？前面提到过，要想和 USB 通信，就得创建一个 **urb**，并且为它赋值，交给 USB Core，它会找到合适的主机控制器，从而进行具体的数据传输。

123 行，为一个 `struct usb_ctrlrequest` 结构体申请了内存，这里又出现了一个新生事物，它在 `include/linux/usb/ch9.h` 文件中定义。

```

140 struct usb_ctrlrequest {
141     __u8 bRequestType;
142     __u8 bRequest;
143     __le16 wValue;
144     __le16 wIndex;
145     __le16 wLength;
146 } __attribute__((packed));

```

这个结构完全对应于 `spec` 里的 Table 9-2，它描述了主机通过控制传输发送给设备的请求 (Device Requests)。主机向设备请求写信息必须得按照协议中规定好的格式，不然设备就会不明白主机是什么意思。

这个结构描述的 **request** 都是在 **SETUP** 包中发送的，**Setup** 包是前面说到的 **Token PID** 类型中的一种，为了更好地理解，这里详细介绍一下控制传输底层的 **packet** 情况。控制传输最少要有两个阶段的 **transaction**：**SETUP** 和 **STATUS**，**SETUP** 和 **STATUS** 中间的那个 **DATA** 阶段是可有可无的。

**transaction** 这个词在很多地方都有，也算是个跨地区跨学科的热门词汇了，在这里称它为事务也好，会话也罢，我还是直呼它的原名 **transaction**，可以理解为主机和设备之间形成的一次完整的交流。

USB 的 **transaction** 可以包括一个 **Token** 包、一个 **Data** 包和一个 **Handshake** 包。

**Token**、**Data** 和 **Handshake** 都属于四种 **PID** 类型，前面提到，如 **SYNC**、**PID**、地址域、**DATA**、**CRC**，并不是所有 **PID** 类型的包都会包括的。**Token** 包只包括 **SYNC**、**PID**、地址域、**CRC**，并没有 **DATA** 字段，它的名字很形象，就是用来标记所在 **transaction** 里接下来动作的，

对于 Out 和 Setup Token 包，里面的地址域指明了接下来要接收 Data 包的端点，对于 In Token 包，地址域指明了接下来哪个端点要发送 Data 包。

还有，只有主机才有权利发送 Token 包，协议中就是这么规定的。别嫌 spec 规定太多，又管这个又管那个的，没有规矩不成方圆。

与 Token 包相比，Data 包中没了地址域，多了 Data 字段，这个 Data 字段对于低速设备最大为 8 字节，对于全速设备最大为 1023 字节，对于高速设备最大为 1024 字节。它就是躲在 Token 后边儿用来传输数据的。Handshake 包的成分就非常简单了，除了 SYNC，它就只包含了一个 PID，通过 PID 取不同的值来报告一个 transaction 的状态，比如数据已经成功接收了等。

控制传输的 SETUP transaction 一般来说有三个阶段，就是主机向设备发送 Setup Token 包，然后发送 Data0 包，如果一切顺利，设备回应 ACK Handshake 包表示没有问题，为什么加上“一般”？如果中间的那个 Data0 包由于某种不可知因素被损坏了，设备就什么都不会回应，这时就成两个阶段了。

SETUP transaction 之后，接下来如果控制传输有 DATA transaction 的话，那就 Data0、Data1 交叉地发送数据包，这是为了实现 data toggle。最后是 STATUS transaction，向主机汇报前面 SETUP 和 DATA 阶段的结果，比如表示主机下达的命令已经完成了，或者主机下达的命令没有完成，或者设备正忙着那没工夫去理会主机的那些命令。

这样经过 SETUP、DATA、STATUS 这三个传输阶段，一个完整的控制传输完成了。主机接下来可以规划下一次的传输。

现在应该可以看出之前说 requests 都在 Setup 包中发送是有问题的，因为 Setup 包本身并没有数据字段，严格来说，它们应该都是在 SETUP transaction 阶段里 Setup 包后的 Data0 包中发送的。

141 行，bRequestType，这个字段别看就一个字节，内容很丰富的，大道理往往都包含在这种小地方。bit7 就表示了控制传输中 DATA transaction 阶段的方向，当然，如果有 DATA 阶段的话。bit5~ bit 6 表示 request 的类型，是标准的 class-specific 的还是 vendor-specific 的。bit0~ bit 4 表示了这个请求针对的是设备、接口，还是端点。内核为它们专门量身定做了一批掩码，也在 ch9.h 文件中。

```

42 /*
43  * USB directions
44  *
45  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
46  * It's also one of three fields in control requests bRequestType.
47  */
48 #define USB_DIR_OUT                0                /* to device */
49 #define USB_DIR_IN                0x80            /* to host */
50
51 /*

```

```

52 * USB types, the second of three bRequestType fields
53 */
54 #define USB_TYPE_MASK                (0x03 << 5)
55 #define USB_TYPE_STANDARD            (0x00 << 5)
56 #define USB_TYPE_CLASS                (0x01 << 5)
57 #define USB_TYPE_VENDOR              (0x02 << 5)
58 #define USB_TYPE_RESERVED            (0x03 << 5)
59
60 /*
61 * USB recipients, the third of three bRequestType fields
62 */
63 #define USB_RECIP_MASK                0x1f
64 #define USB_RECIP_DEVICE              0x00
65 #define USB_RECIP_INTERFACE          0x01
66 #define USB_RECIP_ENDPOINT          0x02
67 #define USB_RECIP_OTHER              0x03
68 /* From Wireless USB 1.0 */
69 #define USB_RECIP_PORT                0x04
70 #define USB_RECIP_RPIPE              0x05

```

142 行，bRequest，表示具体是哪个 request。

143 行，wValue，这个字段是 request 的参数，request 不同，wValue 就不同。

144 行，wIndex，也是 request 的参数，bRequestType 指明 request 针对的是设备上的某个接口或端点时，wIndex 就用来指明是哪个接口或端点。

145 行，wLength，控制传输中 DATA transaction 阶段的长度，方向已经在 bRequestType 那儿指明了。如果这个值为 0，就表示没有 DATA transaction 阶段，bRequestType 的方向位也就无效了。

现在回到 `usb_control_msg` 函数中。很明显要进行控制传输，得首先创建一个 `struct usb_ctrlrequest` 结构体，填上请求的内容。129 行到 133 行就是来使用传递过来的参数初始化这个结构体的。对于刚开始提到的 `SET_ADDRESS` 来说，bRequest 的值就是 `USB_REQ_SET_ADDRESS`，标准请求之一，`ch9.h` 中定义有。因为 `SET_ADDRESS` 请求并不需要 DATA 阶段，所以 wLength 为 0，而且这个请求是针对设备的，所以 wIndex 也为 0。这么一来，bRequestType 的值也只能为 0 了。因为是设置设备地址的，总得把要设置的地址发给设备，不然设备会不知道主机是什么意思，所以请求的参数 wValue 就是之前 Hub 已经指定好的 devnum。其实 `SET_ADDRESS` 请求各个部分的值 spec 9.4.6 里都有规定。

接下来先看 139 行，走到这儿就表示这次通信已经完成了，那么 `struct usb_ctrlrequest` 结构体也就没用了，没用的东西最好精简掉。

## 22. 设备的生命线（三）

137 行，函数 `usb_control_msg` 调用了 `usb_internal_control_msg` 之后就去一边儿睡大觉了，脏活儿累活儿全部留给 `usb_internal_control_msg` 去做了。

```

71 static int usb_internal_control_msg(struct usb_device *usb_dev,
72                                   unsigned int pipe,
73                                   struct usb_ctrlrequest *cmd,
74                                   void *data, int len, int timeout)
75 {
76     struct urb *urb;
77     int retv;
78     int length;
79
80     urb = usb_alloc_urb(0, GFP_NOIO);
81     if (!urb)
82         return -ENOMEM;
83
84     usb_fill_control_urb(urb, usb_dev, pipe, (unsigned char *)cmd, data,
85                         len, usb_api_blocking_completion, NULL);
86
87     retv = usb_start_wait_urb(urb, timeout, &length);
88     if (retv < 0)
89         return retv;
90     else
91         return length;
92 }
```

这个函数粗看过去可以概括为：以一个 `struct urb` 结构体为中心，以 `usb_alloc_urb`、`usb_fill_control_urb`、`usb_start_wait_urb` 三个函数为基本点。

一个中心：`struct urb` 结构体，就是前面多次提到又多次忽略的，只闻其名不见其形的传说中的 `urb`，全称 `usb request block`，USB 通信靠的就是它了。

第一个基本点：`usb_alloc_urb` 函数，创建一个 `urb`，`struct urb` 结构体只能使用它来创建。

第二个基本点：`usb_fill_control_urb` 函数，进行初始化控制 `urb`，`urb` 被创建之后，在使用之前必须要正确被初始化。

第三个基本点：`usb_start_wait_urb` 函数，将 `urb` 提交给 USB Core，以便分配给特定的主机控制器驱动进行处理，然后默默地等待处理结果，或者超时。

```

1126 struct urb
1127 {
1128     /* private: usb core and host controller only fields in the urb */
1129     struct kref kref;           /* reference count of the URB */
1130     spinlock_t lock;           /* lock for the URB */
1131     void *hcpriv;               /* private data for host controller */
1132     atomic_t use_count;         /* concurrent submissions counter */
1133     u8 reject;                  /* submissions will fail */
1134
1135     /* public: documented fields in the urb that can be used by drivers */
```



```

1136     struct list_head urb_list;          /* list head for use by the urb's
1137                                         * current owner */
1138     struct usb_device *dev; /* (in) pointer to associated device */
1139     unsigned int pipe;          /* (in) pipe information */
1140     int status;                /* (return) non-ISO status */
1141     unsigned int transfer_flags; /* (in) URB_SHORT_NOT_OK | ...*/
1142     void *transfer_buffer;      /* (in) associated data buffer */
1143     dma_addr_t transfer_dma; /* (in) dma addr for transfer_buffer */
1144     int transfer_buffer_length; /* (in) data buffer length */
1145     int actual_length;          /* (return) actual transfer length */
1146     unsigned char *setup_packet; /* (in) setup packet (control only) */
1147     dma_addr_t setup_dma;      /* (in) dma addr for setup_packet */
1148     int start_frame;           /* (modify) start frame (ISO) */
1149     int number_of_packets;      /* (in) number of ISO packets */
1150     int interval;              /* (modify) transfer interval
1151                                 * (INT/ISO) */
1152     int error_count;           /* (return) number of ISO errors */
1153     void *context;             /* (in) context for completion */
1154     usb_complete_t complete;    /* (in) completion routine */
1155     struct usb_iso_packet_descriptor iso_frame_desc[0];
1156                                 /* (in) ISO ONLY */
1157 };

```

1129 行，`kref`，`urb` 的引用计数。别看它是隐藏在 `urb` 内部的一个不起眼的小角色，但小角色做大事情，它决定了一个 `urb` 的生死存亡。一个 `urb` 有用没用，是继续委以重任还是无情销毁都要看它的脸色。那为什么 `urb` 的生死要掌握在这个小小的引用计数手里？

前面说过，主机与设备之间通过管道来传输数据，管道的一端是主机上的一个缓冲区，另一端是设备上的端点。管道之中流动的数据，在主机控制器和设备看来是一个个 `packets`，在咱们看来就是 `urb`。因而，端点之中就有一个叫 `urb` 的队列。不过，这并不代表一个 `urb` 只能发配给一个端点，它可能通过不同的管道发配给不同的端点，那么这样一来，我们如何知道这个 `urb` 正在被多少个端点使用，如何判断这个 `urb` 的生命已经结束？如果没有任何一个端点在使用它，而我们又无法判断这种情况，它就会永远地飘荡在 USB 的世界里。我们需要寻求某种办法在这种情况下给它们一个好的归宿，这就是引用计数。每多一个使用者，它的引用计数就加 1，每减少一个使用者，引用计数就减 1，如果连最后一个使用者都释放了这个 `urb`，宣称不再使用它了，那它的生命周期就走到了尽头，会自动销毁。

如何来表示这个神奇的引用计数？其实它是一个 `struct kref` 结构体，在 `include/linux/kref.h` 中定义。

```

23 struct kref {
24     atomic_t refcount;
25 };

```

这个结构与 `struct urb` 相比简约到极点了。不过别看它简单，内核中就是使用它来判断一个对象有没有用。它里边儿只包括了一个原子变量，为什么是原子变量？既然都使用引用计数了，那就说明可能同时有多个地方在使用这个对象，总要考虑一下它们同时修改这个计数的可能性吧，也就是俗称的并发访问，那怎么办？加一个锁？就这么一个整数值专门加个一锁未免也大

材小用了，所以就使用了原子变量。围绕这个结构，内核中还定义了几个专门操作引用计数的函数，它们在 `lib/kref.c` 中定义。

```

21 void kref_init(struct kref *kref)
22 {
23     atomic_set(&kref->refcount,1);
24     smp_mb();
25 }
26
31 void kref_get(struct kref *kref)
32 {
33     WARN_ON(!atomic_read(&kref->refcount));
34     atomic_inc(&kref->refcount);
35     smp_mb__after_atomic_inc();
36 }
37
52 int kref_put(struct kref *kref, void (*release)(struct kref *kref))
53 {
54     WARN_ON(release == NULL);
55     WARN_ON(release == (void (*)(struct kref *))kfree);
56
57     if (atomic_dec_and_test(&kref->refcount)) {
58         release(kref);
59         return 1;
60     }
61     return 0;
62 }

```

整个 `kref.c` 文件就定义了这么三个函数，`kref_init` 初始化，`kref_get` 将引用计数加 1，`kref_put` 将引用计数减 1 并判断是不是为 0，如果为 0 的话就调用参数中 `release` 函数指针指向的函数把对象销毁掉。它们对 `refcount` 的操作都是通过原子变量特有的操作函数，原子变量当然要使用专门的操作函数了，编译器还能做一些优化，否则直接使用一般的变量就可以了。如果你直接像对待一般整型值一样对待它，编译器也会看不过去你的行为，直接给你一个 `error`。

提醒一下，`kref_init` 初始化时，是把 `refcount` 的值初始化为 1，而不是 0。还有一点要说的是 `kref_put` 参数中的那个函数指针，你不能传递一个 `NULL` 过去，否则这个引用计数就只是计数，而背离了最初的目的。要记住，我们需要在这个计数减到为 0 时将销毁嵌入这个引用计数 `struct kref` 结构体的对象，所以这个函数指针也不能为 `kfree`，因为这样就只是把这个 `struct kref` 结构体给销毁了，而不是整个对象。

第三个问题，如何使用 `struct kref` 结构来为我们的对象计数？当然我们需要把这样一个结构嵌入到你希望计数的对象里，不然你根本无法对对象在它整个生命周期里的使用情况做出判断。但是我们是几乎见不到内核中直接使用上面那几个函数来给对象计数的，而是每种对象又定义了自己专用的引用计数函数，比如这里的 `urb`，在 `drivers/usb/core/urb.c` 中定义。

```

31 void usb_init_urb(struct urb *urb)
32 {
33     if (urb) {
34         memset(urb, 0, sizeof(*urb));
35         kref_init(&urb->kref);

```

```

36         spin_lock_init(&urb->lock);
37     }
38 }
39
81 void usb_free_urb(struct urb *urb)
82 {
83     if (urb)
84         kref_put(&urb->kref, urb_destroy);
85 }
86
97 struct urb * usb_get_urb(struct urb *urb)
98 {
99     if (urb)
100         kref_get(&urb->kref);
101     return urb;
102 }

```

usb\_init\_urb, usb\_get\_urb, usb\_free\_urb 这三个函数分别调用了前面的 struct kref 结构的三个操作函数来进行引用计数的初始化、加 1、减 1。什么叫封装？这就叫封装。usb\_free\_urb 里给 kref\_put 传递的那个函数 urb\_destroy，它也在 urb.c 中定义。

```

9 #define to_urb(d) container_of(d, struct urb, kref)
10
11 static void urb_destroy(struct kref *kref)
12 {
13     struct urb *urb = to_urb(kref);
14     kfree(urb);
15 }

```

这个 urb\_destroy 函数首先调用了 to\_urb，实际上就是一个 container\_of 来获得引用计数关联的 urb，然后使用 kfree 函数将它销毁。

回到函数 1130 行，lock，一把自旋锁。每个 urb 都有一把自旋锁。

1131 行，hcpriv，走到今天，你应该明白这个 urb 最终还是要提交给主机控制器驱动的，这个字段就是 urb 里主机控制器驱动的自留地。

1132 行，use\_count，这里又是一个计数，不过此计数非彼计数，它与上面那个用来追踪 urb 生命周期的 kref 一点儿关系也没有。那它是用来做什么的？

先了解一下使用 urb 来完成一次完整的 USB 通信都要经历哪些阶段。首先，驱动程序发现自己有与 USB 设备通信的需要，于是创建一个 urb，并指定它的目的地是设备上的哪个端点，然后提交给 USB Core，USB Core 将它修修补补进行一些美化之后再移交给主机控制器的驱动程序 HCD，HCD 会去解析这个 urb，了解它的目的是什么，并与 USB 设备进行相应的交流，在交流结束，urb 的目的达到之后，HCD 再把这个 urb 的所有权移交回驱动程序。

这里的 use\_count 就是在 USB Core 将 urb 移交给 HCD，办理移交手续时，插上了一脚，每当走到这一步，它的值就会加 1。什么时候减 1？在 HCD 重新将 urb 的所有权移交回驱动程序时。只要 HCD 拥有这个 urb 的所有权，那么此 urb 的 use\_count 就不会为 0。这么一说，似乎

`use_count` 也有一点追踪 `urb` 生命周期的味道了，当它的值大于 0 时，就表示当前有 HCD 正在处理它，和上面的 `kref` 概念上有部分的重叠，不过，显然它们之间是有区别的，没区别的话，这里干吗要用两个计数？

上面的那个 `kref` 实现方式是内核中统一的引用计数机制，当计数减为 0 时，`urb` 对象就被 `urb_destroy` 给销毁了。这里的 `use_count` 只是用来统计当前这个 `urb` 是不是正在被哪个 HCD 处理，即使它的值为 0，也只是说明没有 HCD 在使用它而已，并不代表就得把它给销毁掉。比方说，HCD 利用完了 `urb`，把它还给了驱动，这时驱动还可以对这个 `urb` 进行检修，再提交给哪个 HCD 去使用。

下面的问题就是既然它不会平白无故地多出来，那它究竟是用来干什么的？还要从刚提到的那几个阶段说起。`urb` 驱动也创建了，该提交的也提交了，HCD 正处理着，可驱动反悔了，它不想再继续这次通信了，想将这个 `urb` 给终止掉，善解人意的 USB Core 当然会给驱动提供这样的接口来满足这样的需要。不过这个需要被细分为两种方式，一种是驱动只想通过 USB Core 告诉 HCD 这个 `urb` 想终止掉，不用再处理了，然后它不想在那里等着 HCD 的处理，想忙别的事去，这就是俗称的“异步”，对应的函数是 `usb_unlink_urb`。当然对应的还有另一种同步，驱动会在那里苦苦等候着 HCD 的处理结果，等待着 `urb` 被终止，对应的函数是 `usb_kill_urb`。而 HCD 将这次通信终止后，同样会将 `urb` 的所有权移交回驱动。那么驱动通过什么判断 HCD 已经终止了这次通信？就是通过这里的 `use_count`，驱动会在 `usb_kill_urb` 里面一直等待着这个值变为 0。

1133 行，`reject`，拒绝，拒绝什么？又是被谁拒绝？

在目前版本的内核中，只有 `usb_kill_urb` 函数有特权对它进行修改，那么显然 `reject` 就与上面说的 `urb` 终止有关了。那就看一看 `drivers/usb/core/urb.c` 中定义的这个函数。

```

464 void usb_kill_urb(struct urb *urb)
465 {
466     might_sleep();
467     if (!(urb && urb->dev && urb->dev->bus))
468         return;
469     spin_lock_irq(&urb->lock);
470     ++urb->reject;
471     spin_unlock_irq(&urb->lock);
472
473     usb_hcd_unlink_urb(urb, -ENOENT);
474     wait_event(usb_kill_urb_queue, atomic_read(&urb->use_count) == 0);
475
476     spin_lock_irq(&urb->lock);
477     --urb->reject;
478     spin_unlock_irq(&urb->lock);
479 }
```

466 行，因为 `usb_kill_urb` 函数要一直等候着 HCD 将 `urb` 终止掉，它必须是可以休眠的。所以说 `usb_kill_urb` 函数不能用于中断上下文，它必须能够休眠将自己占的资源给让出来。

`might_sleep` 函数，用来判断一下 `usb_kill_urb` 函数是不是处在能够休眠的情况，如果不是，就会打印出一大堆的堆栈信息，比如你在中断上下文调用了这个函数时。不过，它也就是基于调试的目的用一用，方便日后找错，并不能强制哪个函数改变自己的上下文。

467 行，这里就是判断一下 `urb` 要去的那个设备，还有那个设备现在的总线有没有，如果不存在，还是返回吧。

469 行，去获得每个 `urb` 都有的那把锁，然后将 `reject` 加 1。加 1 有什么用？其实目前版本的内核中只有两个地方用到了这个值进行判断。第一个地方是在 USB Core 将 `urb` 提交给 HCD，正在办移交手续时，如果 `reject` 大于 0，就不再接着移交了，也就是说这个 `urb` 被 HCD 给拒绝了。这是为了防止这边儿正在终止这个 `urb`，那边儿的某个地方却又妄想将这个 `urb` 重新提交给 HCD。

473 行，这里告诉 HCD 驱动要终止这个 `urb` 了，`usb_hcd_unlink_urb` 函数也只是告诉 HCD 一声，然后不管 HCD 怎么处理就返回了。

474 行，上面的 `usb_hcd_unlink_urb` 是返回了，但并不代表 HCD 已经将 `urb` 给终止了，HCD 可能没那么快，所以这里 `usb_kill_urb` 要休息一下，等人通知它。这里使用了 `wait_event` 这个宏来实现休眠，`usb_kill_urb_queue` 是在 `/drivers/usb/core/hcd.h` 中定义的一个等待队列，专门给 `usb_kill_urb` 休息用的。需要注意的是，这里的唤醒条件 `use_count` 必须等于 0。

在哪里能唤醒正在睡大觉的 `usb_kill_urb`？这牵扯到了第二个使用 `reject` 来做判断的地方。在 HCD 将 `urb` 的所有权返还给驱动时，会对 `reject` 进行判断，如果 `reject` 大于 0，就调用 `wake_up` 唤醒在 `usb_kill_urb_queue` 上休息的 `usb_kill_urb`。这也好理解，HCD 都要将 `urb` 的所有权返回给驱动了，那当然就是已经处理完了，放在这里就是已经将这个 `urb` 终止了，`usb_kill_urb` 等的就是这一天的到来，当然就要醒过来继续往下走了。

476 行，再次获得 `urb` 的那把锁，将 `reject` 刚才增加的 1 减掉。`urb` 都已经终止了，也没人再去拒绝它了。

与 `usb_kill_urb` 相比，`usb_unlink_urb` 函数就简单多了。

```
435 int usb_unlink_urb(struct urb *urb)
436 {
437     if (!urb)
438         return -EINVAL;
439     if (!(urb->dev && urb->dev->bus))
440         return -ENODEV;
441     return usb_hcd_unlink_urb(urb, -ECONNRESET);
442 }
```

`usb_unlink_urb` 函数只是把自己的意愿告诉 HCD，然后就返回了。

`struct urb` 结构中的前面这几个函数，只是 USB Core 和主机控制器驱动需要关心的，实际

的驱动里根本用不着也管不着，它们就是 USB 和 HCD 的后花园，想种点什么不种什么都由写这块儿代码的人决定，它们在里面怎么为所欲为都不关写驱动的人什么事。USB 在 linux 里起起伏伏这么多年，前边儿的这些内容早就变过多少次，说不定你今天还能看到谁，到接下来的哪天就看不到了，不过，变化的是形式，不变的是道理。

而驱动要做的只是创建一个 urb，然后初始化，再把它提交给 USB Core 就可以了，使用不使用引用计数，加不加锁之类的事都不用去操心。感谢 David Brownell，感谢 Alan Stern，感谢……没有他们就没有 USB 在 Linux 里的今天。

## 23. 设备的生命线（四）

在 struct urb 中，就是每个写 usb 驱动的人都需要关心的了，坐这儿看了半天，struct urb 才露出来这么一个角。

1136 行，urb\_list，还记得每个端点都会有的那个 urb 队列吗？那个队列就是由这里的 urb\_list 一个一个地链接起来的。HCD 每收到一个 urb，就会将它添加到这个 urb 指定的那个端点的 urb 队列里去。这个链表的头儿在哪儿？当然是在端点里，就是端点里的那个 struct list\_head 结构体成员。

1138 行，dev，它表示 urb 要去的那个 USB 设备。

1139 行，pipe，urb 到达端点之前，需要经过一个通往端点的管道，就是这个 pipe。怎么表示一个管道？管道有两端，一端是主机上的缓冲区，一端是设备上的端点。既然有两端，总要有个方向吧！前面说过，端点有四种类型，那么与端点相生相依的管道也应该不只一种。

这么说来，确定一条管道至少要知道管道两端的地址、方向和类型，不过这两端里主机是确定的，需要确定的只是另一端设备的地址和端点的地址。怎么将这些内容组合起来表示成一个管道？一个包含了各种成员属性的结构再加上一些操作函数？多么完美的封装，但是不需要这么做，一个整型值再加上一些宏就够了。

先看一看管道，也就是这个整型值的构成，bit7 用来表示方向，bit8~ bit14 表示设备地址，bit15~ bit18 表示端点号，早先说过，设备地址用 7 位来表示，端点号用 4 位来表示，剩下的 bit30~ bit31 表示管道类型。再看一看围绕管道的一些宏，在 include/linux/usb.h 中定义。

```
1407 #define PIPE_ISOCHRONOUS          0
1408 #define PIPE_INTERRUPT              1
1409 #define PIPE_CONTROL                2
1410 #define PIPE_BULK                   3
1411
1412 #define usb_pipein(pipe)            ((pipe) & USB_DIR_IN)
```

```
1413 #define usb_pipeout(pipe)      (!usb_pipein(pipe))
1414
1415 #define usb_pipedevicex(pipe)   (((pipe) >> 8) & 0x7f)
1416 #define usb_pipeendpoint(pipe) (((pipe) >> 15) & 0xf)
1417
1418 #define usb_pipetype(pipe)      (((pipe) >> 30) & 3)
1419 #define usb_pipeisoc(pipe)      (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
1420 #define usb_pipeint(pipe)       (usb_pipetype((pipe)) == PIPE_INTERRUPT)
1421 #define usb_pipecontrol(pipe)   (usb_pipetype((pipe)) == PIPE_CONTROL)
1422 #define usb_pipebulk(pipe)      (usb_pipetype((pipe)) == PIPE_BULK)
```

现在来看一下，如何创建一个管道？主机和设备要交流必须通过管道，你必须得创建一个管道给 `urb`，它才知道路怎么走。不过在说怎么创建一个管道前，先说一个有关管道的故事。

1801 年，在意大利中部的小山村，有两个名叫柏波罗和布鲁诺的年轻人，他们的最大梦想是成为村子里最富有的人。有一天，喜鹊在枝头唧唧喳喳地叫，他们的好运也就随着来了。村里决定雇两个人把附近河里的水运到村广场的水缸里去，他们得到了这个机会。“我提一桶水，只收他一分钱，我提 10 桶水赚 1 毛钱，我一天提一百桶水！能赚多少钱啊！”布鲁诺激动地盘算着。但柏波罗却想着一天才几分钱的报酬，还要这样来回提水，还不如干脆修一条管道将水从河里引到村里去。于是布鲁诺每天辛勤地提着水，很快买了新衣服，买了驴，虽然仍然买不起车也买不起房，但已经被看做是中产阶级了，而柏波罗每天还要抽出一部分提水的时间去挖管道，收入是入不敷出，挖管道的同时还要接收很多人对他的嘲笑。两年后，柏波罗的管道完工了，水哗哗地直往村里流，钱哗哗地直往口袋里钻，而此时布鲁诺因为长时间的提桶工作变得腰弯背驼。柏波罗成了奇迹的创造者，他没有被赞扬冲昏头脑，他想的是创建更多的管道，他邀请布鲁诺加入了他的管道事业，从此他们的管道事业芝麻开花节节高，遍布了全球。

这个故事告诉我们，管道很重要。显然，写代码的人也深刻地认识到了这个道理，于是内核的 `include/linux/usb.h` 文件中多了很多专门用来创建不同管道的宏。

```
1432 static inline unsigned int __create_pipe(struct usb_device *dev,
1433     unsigned int endpoint)
1434 {
1435     return (dev->devnum << 8) | (endpoint << 15);
1436 }
1437
1438 /* Create various pipes... */
1439 #define usb_sndctrlpipe(dev, endpoint) \
1440     ((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint))
1441 #define usb_rcvctrlpipe(dev, endpoint) \
1442     ((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
1443 #define usb_sndisocpipe(dev, endpoint) \
1444     ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint))
1445 #define usb_rcvisocpipe(dev, endpoint) \
1446     ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
1447 #define usb_sndbulkpipe(dev, endpoint) \
1448     ((PIPE_BULK << 30) | __create_pipe(dev, endpoint))
1449 #define usb_rcvbulkpipe(dev, endpoint) \
1450     ((PIPE_BULK << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
1451 #define usb_sndintpipe(dev, endpoint) \
1452     ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint))
```

```

1453 #define usb_rcvintpipe(dev,endpoint) \
1454     ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)

```

端点有四种，对应着管道也就有四种，同时端点是有 IN 也有 OUT 的，相应的管道也就有两个方向，于是，上面就出现了八个创建管道的宏。有了 `struct usb_device` 结构体，也就是说知道了设备地址，再加上端点号，就可以需要什么管道就创建什么管道了。`__create_pipe` 宏只是一个幕后的角色，用来将设备地址和端点号放在管道正确的位置上。

1140 行，`status`，`urb` 的当前状态。`urb` 当然是可以有多种状态的，`urb` 当前什么状态就是让我们了解出了什么事情。

1141 行，`transfer_flags`，一些标记，可用的值都在 `include/linux/usb.h` 里有定义。

```

942 #define URB_SHORT_NOT_OK          0x0001 /* report short reads as errors */
943 #define URB_ISO_ASAP              0x0002 /* iso-only, urb->start_frame
944                                     * ignored */
945 #define URB_NO_TRANSFER_DMA_MAP 0x0004 /* urb->transfer_dma valid on submit
*/
946 #define URB_NO_SETUP_DMA_MAP      0x0008 /*urb->setup_dma valid on submit*/
947 #define URB_NO_FSBR               0x0020 /* UHCI-specific */
948 #define URB_ZERO_PACKET           0x0040 /* Finish bulk OUT with short packet */
949 #define URB_NO_INTERRUPT          0x0080 /* HINT: no non-error interrupt
950                                     * needed */

```

`URB_SHORT_NOT_OK`，这个标记只对用来从 IN 端点读取数据的 `urb` 有效，意思是如果从一个 IN 端点那里读取了一个比较短的数据包，就可以认为是错误的。那么这里的 `short` 究竟短到什么程度？

之前说到端点时，就知道端点描述符里有一个叫 `wMaxPacketSize`，指明了端点一次能够处理的最大字节数。另外前面也提了，在 USB 的世界里是有很多种包的，四种 PID 类型，每种 PID 下边儿还有一些细分的品种。这四种 PID 里面，有一个叫 `Data` 的，也只有它里边儿有一个数据字段，像其他的 `Token`、`Handshake` 之类的 PID 类型都是没有这个字段的，所以里里外外看过去，还只有 `Data` PID 类型的包最实在，就是用来传输数据的，但是它里面并不是只有一个数据字段，还有 `SYNC`、`PID`、地址域、`CRC` 等陪伴在数据字段的左右。

现在又有一个问题出来了，每个端点描述符里的 `wMaxPacketSize` 所表示的最大字节数都包括了哪些部分？是整个 `packet` 的长度吗？我可以负责任地告诉你，它只包括了 `Data` 包中面数据字段，俗称 `data payload`，其他的数据字段都是协议本身需要的信息，和 `TCP/IP` 里的报头差不多。

`wMaxPacketSize` 与 `short` 有什么关系？关系还不小，`short` 是与 `wMaxPacketSize` 相比的，如果从 IN 端点那儿收到了一个比 `wMaxPacketSize` 要短的包，同时也设置了 `URB_SHORT_NOT_OK` 这个标志，那么就可以认为传输出错了。

本来如果收到一个比较短的包是意味着这次传输到此为止就结束了，你想想，`data`



payload 的长度最大必须为 `wMaxPacketSize`，这个规定是不可违背的，但是如果端点想给你的数据不止这么多，怎么办？这就需要分成多个 `wMaxPacketSize` 大小的 data payload 来传输，有时数据不会那么凑巧刚好能平分成多个整份。这时，最后一个 data payload 的长度就会比 `wMaxPacketSize` 小，这种情况本来意味着端点已经传完了它想传的数据，释放完了自己的需求，这次传输就该结束了，不过如果你设置了 `URB_SHORT_NOT_OK` 标志，HCD 这边就会认为发生了错误。

`URB_ISO_ASAP`，这个标志只是为了方便等时传输使用。等时传输和中断传输在 spec 里都被认为是 `periodic transfers`，也就是周期传输，我们都知道在 USB 的世界里都是主机占主导地位，设备是没多少发言权的，但是对于等时传输和中断传输，端点可以对主机表达自己一种美好的期望，希望主机能够隔多长时间访问自己一次，这个期望的时间就是这里说的周期。

当然，期望与现实是有一段距离的，如果期望的都能成为现实，我们还研究 USB 干什么，端点的这个期望能不能得到满足，要看主机控制器答应不答应。

对于等时传输，一般来说也就一帧（微帧）一次，主机也很忙，再多也抽不出空儿来。那么如果你有一个用于等时传输的 urb，提交给 HCD 时，就得告诉 HCD 它应该从哪一帧开始的，就要对下面要讲到的 `start_frame` 赋值，也就是说告诉 HCD 等时传输开始的那一帧（微帧）的帧号，如果你留心，应该还会记得前面说过在每帧或微帧（Microframe）的开始都会有一个 SOF Token 包，这个包中就含有一个帧号字段，记录了那一帧的编号。

这样的话，一是要去设置这个 `start_frame`，二是到你设置的那一帧时，如果主机控制器没空儿开始等时传输，怎么办？于是，就出现了 `URB_ISO_ASAP`，它的意思就是告诉 HCD 什么时候不忙就什么时候开始，就不用指定开始的帧号了。所以说，你如果想进行等时传输，又不想标新立异的话，就还是设置 `start_frame` 吧。

`URB_NO_TRANSFER_DMA_MAP`，还有 `URB_NO_SETUP_DMA_MAP`，这两个标志都是与 DMA 有关的。什么是 DMA？DMA 就是“外设”，比如 USB 摄像头，和内存之间直接进行数据交换，不用通过 CPU。本来，在我们的计算机里，CPU 自认为是老大，什么事都要去插一脚，都要经过它去协调处理。可是这样的话就影响了数据传输的速度，有 DMA 和没有 DMA 区别就是这么大。

USB 的世界里也是要与时俱进，所以 DMA 也是少不了的。一般来说，都是驱动里提供了 `kmalloc` 等分配的缓冲区，HCD 进行一定的 DMA 映射处理，DMA 映射是干什么的？外设和内存之间进行数据交换总要互相认识吧，外设是通过各种总线连到主机里面的，使用的是总线地址，而内存使用的是虚拟地址，它们之间本来就是两条互不相交的平行线，要让它们中间产生连接点，必须得将一个地址转化为另一个地址，这样才能找得到对方，才能互通有无，而 DMA 映射就是起这个作用的。

这只是粗略说法，实际上即使千言万语也道不完的。DMA 映射可是高技术含量的活儿，

所以在某些平台上非常费时费力。为了分担 HCD 的压力，于是就有了这里的两个标志，告诉 HCD 不要再自己做 DMA 映射了，驱动提供的 urb 里已经提供有 DMA 缓冲区地址。具体提供了哪些 DMA 缓冲区？就涉及下面的 `transfer_buffer`, `transfer_dma`, 还有 `setup_packet`, `setup_dma` 这两个函数了。

`URB_NO_FSB`，这是给 UHCI 用的函数。

`URB_ZERO_PACKET`，这个标志表示批量的 OUT 传输必须使用一个 short packet 来结束。批量传输的数据大于批量端点的 `wMaxPacketSize` 时，需要分成多个 Data 包来传输，最后一个 data payload 的长度可能等于 `wMaxPacketSize`，也可能小于 `wMaxPacketSize`，当等于 `wMaxPacketSize` 时，如果同时设置了 `URB_ZERO_PACKET` 标志，就需要再发送一个长度为 0 的数据包来结束这次传输，如果小于 `wMaxPacketSize` 就没必要多此一举了。如果你要问，当批量传输的数据小于 `wMaxPacketSize` 时怎么办？也没必要再发送 0 长的数据包，因为此时发送的这个数据包本身就是一个 short packet。

`URB_NO_INTERRUPT`，这个标志用来告诉 HCD，在 URB 完成后，不要请求一个硬件中断，当然这就意味着你的结束处理函数可能不会在 urb 完成后立即被调用，而是在之后的某个时间被调用，USB Core 会保证为每个 urb 调用一次结束处理函数。

还是回到 `struct urb`，1142 行到 1144 行，`transfer_buffer`, `transfer_dma`, `transfer_buffer_length`，前面说过管道的一端是主机上的缓冲区，另一端是设备上的端点，这三个家伙就是描述主机上的缓冲区的函数。

`transfer_buffer` 是使用 `kmalloc` 分配的缓冲区，`transfer_dma` 是使用 `usb_buffer_alloc` 分配的 dma 缓冲区。HCD 不会同时使用它们两个，如果 urb 自带了 `transfer_dma`，就要同时设置 `URB_NO_TRANSFER_DMA_MAP` 来告诉 HCD 一声，不用它再费心做 DMA 映射了。`transfer_buffer` 是必须要设置的，因为不是所有的主机控制器都能够使用 DMA 的，万一遇到这样的情况，也好有一个备用。`transfer_buffer_length` 指的就是 `transfer_buffer` 或 `transfer_dma` 的长度。

1145 行，`actual_length`，urb 结束之后，会用这个字段告诉你实际上传输了多少数据。

1146 行到 1147 行，`setup_packet`, `setup_dma`，同样是两个缓冲区，一个是 `kmalloc` 分配的，一个是用 `usb_buffer_alloc` 分配的，不过，这两个缓冲区是控制传输专用的，记得 `struct usb_ctrlrequest` 吗？它们保存的就是一个 `struct usb_ctrlrequest` 结构体，如果你的 urb 设置了 `setup_dma`，同样要设置 `URB_NO_SETUP_DMA_MAP` 标志来告诉 HCD。如果进行的是控制传输，`setup_packet` 是必须要设置的，也是为了防止出现主机控制器不能使用 DMA 的情况。

1148 行，`start_frame`，如果你没有指定 `URB_ISO_ASAP` 标志，就必须自己设置 `start_frame`，指定等时传输在哪个帧或哪个微帧开始。如果指定了 `URB_ISO_ASAP`，urb 结束时会使用这个

值返回实际的开始帧号。

1150 行, `interval`, 等时传输和中断传输专用。`Interval` 是间隔时间的意思。什么的间隔时间? 就是上面说的端点希望主机轮询自己的时间间隔。这个值和端点描述符里的 `bInterval` 是一样的, 你不能随便地指定一个, 然后就去春秋大梦, 以为到时间了梦里的名车美女都会跑出来, 协议中对你指定的值是有范围限制的。对于中断传输, 全速时, 这个范围为  $1\text{ ms} \sim 255\text{ ms}$ , 低速时为  $10\text{ ms} \sim 255\text{ ms}$ , 高速时为  $1\text{ ms} \sim 16\text{ ms}$ , 这个  $1\text{ ms} \sim 16\text{ ms}$  只是 `bInterval` 可以取的值, 实际的间隔时间为 2 的  $(\text{bInterval}-1)$  次方乘以  $125\text{ ms}$ , 也就是 2 的  $(\text{bInterval}-1)$  次方个微帧。

对于等时传输, 没有低速等时传输根本就不是低速端点负担得起的, 对于全速和高速, 这个范围也是为  $1\text{ ms} \sim 16\text{ ms}$ , 间隔时间由 2 的  $(\text{bInterval}-1)$  次方算出来, 单位为帧或微帧。

这样看来, 每一帧或微帧里, 你最多只能期望有一次等时传输和中断传输, 不能再多了。

不过即使完全按照上面的范围来取, 你的期望也并不是就肯定可以实现的, 因为对于高速传输来说, 最多有 80% 的总线时间用于这两种传输, 对于低速传输和全速传输要多一点, 达到 90%。这个时间的分配都由主机控制器掌握着, 所以你的期望能不能实现还要看主机控制器的脸色, 没办法, 它就有这种权力。

1153 行, `context`, 驱动设置了给下面的结束处理函数用的。比如可以将自己驱动里描述自己设备的结构体放在里面, 在结束处理函数中就可以取出来。

1154 行, `complete`, 一个指向结束处理函数的指针, 传输成功完成, 或者中间发生错误时就会调用它, 驱动可以在这里面检查 `urb` 的状态, 并做一些处理。比如可以释放这个 `urb`, 或者重新提交给 HCD。比如说摄像头吧, 你向 HCD 提交了一个等时的 `urb`, 从摄像头那里读取视频数据, 传输完成时调用了你指定的这个结束处理函数, 并在里面取出了 `urb` 里面获得的数据进行解码等处理, 然后怎么办? 总不会这一个 `urb` 读取的数据就够了吧, 所以需要获得更多的数据, 那你也总不会再去创建、初始化一个等时的 `urb` 吧, 很明显刚刚的那个 `urb` 可以继续用, 只要将它再次提交给 HCD 就可以了。这个函数指针的定义在 `include/linux/usb.h`。

```
961 typedef void (*usb_complete_t)(struct urb *);
```

还有三个函数, 都是等时传输专用的, 等时传输与其他传输不一样, 可以指定传输多少个 packet, 每个 packet 使用 `struct usb_iso_packet_descriptor` 结构来描述。1155 行的 `iso_frame_desc` 就表示了一个变长的 `struct usb_iso_packet_descriptor` 结构体数组, 而 1149 行的 `number_of_packets` 指定了这个结构体数组的大小, 也就是要传输多少个 packet。

要说明的是, 这里说的 packet 不是说在一次等时传输里传输了多个 Data packet, 而是说在一个 `urb` 里指定了多次的等时传输, 每个 `struct usb_iso_packet_descriptor` 结构体都代表了一次等时传输。

这里对等时传输底层的 packet 情况说明一下。不像控制传输最少要有 SETUP 和 STATUS 两个阶段的 transaction，等时传输只有 Isochronous transaction，即等时 transaction 一个阶段，一次等时传输就是一次等时 transaction 的过程。

而等时 transaction 也只有两个阶段，就是主机向设备发送 OUT Token 包，然后发送一个 Data 包，或者是主机向设备发送 IN Token 包，然后设备向主机发送一个 Data 包，这个 Data 包中 data payload 的长度只能小于或者等于等时端点的 wMaxPacketSize 值。

这里没有了 Handshake 包，是因为不需要，等时传输是不保证数据完全正确无误到达的，没有什么错误重传机制，也就不需要使用 Handshake 包来汇报。对它来说实时性要比正确性重要得多，你的摄像头一秒钟少给你一帧和多给你一帧没有本质的区别，如果等时传输延迟几秒，感觉就明显不同了。

所以对于等时传输来说，在完成了 number\_of\_packets 次传输之后，会去调用你的结束处理函数，在里面对数据做处理，而 1152 行的 error\_count 记录了这么多次传输中发生错误的次数。

现在看一看 struct usb\_iso\_packet\_descriptor 结构的定义，在 include/linux/usb.h 中定义。

```

952 struct usb_iso_packet_descriptor {
953     unsigned int offset;
954     unsigned int length;           /* expected length */
955     unsigned int actual_length;
956     int status;
957 };

```

offset 表示 transfer\_buffer 里的偏移位置，你不是指定了要进行 number\_of\_packets 次等时传输吗，那么也要准备够这么多次传输用的缓冲区，当然不是说让你准备多个缓冲区。没必要，都放 transfer\_buffer 或者 transfer\_dma 里面好了，只要记着每次传输对应的数据偏移就可以。length 是预期的这次等时传输 Data 包中数据的长度，注意这里说的是预期，因为实际传输时因为各种原因可能不会有那么多数据，urb 结束时，每个 struct usb\_iso\_packet\_descriptor 结构体的 actual\_length 就表示了各次等时传输实际传输的数据长度，而 status 分别记录了它们的状态。

## 24. 设备的生命线（五）

下面接着看内核代码的三个基本点。

第一个基本点，usb\_alloc\_urb 函数，创建 urb 的专用函数，为一个 urb 申请内存并做初始化，在 drivers/usb/core/urb.c 中定义。

```

56 struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
57 {

```

```

58     struct urb *urb;
59
60     urb = kmalloc(sizeof(struct urb) +
61                 iso_packets * sizeof(struct usb_iso_packet_descriptor),
62                 mem_flags);
63     if (!urb) {
64         err("alloc_urb: kmalloc failed");
65         return NULL;
66     }
67     usb_init_urb(urb);
68     return urb;
69 }

```

这函数只做了两件事情，拿 `kmalloc` 来为 `urb` 申请内存，然后调用 `usb_init_urb` 进行初始化。`usb_init_urb` 函数在前面讲 `struct urb` 中的引用计数时已经讲过了，它主要的作用就是初始化 `urb` 的引用计数，还用 `memset` 顺便把这里给 `urb` 申请的内存清零。

`usb_alloc_urb` 说：“别看我简单，我也是很有内涵的。”先看第一个问题，它的第一个参数 `iso_packets`，表示 `struct urb` 结构最后那个变长数组 `iso_frame_desc` 的元素数目，也就是应该与 `number_of_packets` 的值相同，所以对于控制/中断/批量传输，这个参数都应该为 0。这也算是给我们示范了一下变长数组的用法。

第二个问题是参数 `mem_flags` 的类型 `gfp_t`，早几个版本的内核，类型还是 `int`，当然这里变成 `gfp_t` 是因为 `kmalloc` 参数中的标志参数的类型从 `int` 变成 `gfp_t` 了，你要用 `kmalloc` 来申请内存，就得遵守它的规则。不过这里要说的不是 `kmalloc`，而是 `gfp_t`，它在江湖上也没出现多久，名号还没打出来，很多人不了解，我们来调查一下它的背景。它在 `include/linux/types.h` 中定义。

```

193 typedef unsigned __bitwise__ gfp_t;

```

很显然，要了解 `gfp_t`，关键是要了解 `__bitwise__`，它也在 `types.h` 中定义。

```

170 #ifdef __CHECKER__
171 #define __bitwise__ __attribute__((bitwise))
172 #else
173 #define __bitwise__
174 #endif

```

`__bitwise__` 的含义又取决于是否定义了 `__CHECKER__`，如果没有定义 `__CHECKER__`，那 `__bitwise__` 就什么也不是。在哪里定义了 `__CHECKER__`？内核代码里就没有哪个地方定义了 `__CHECKER__`，它是有关 `Sparse` 工具的，内核编译时的参数决定了是不是使用 `Sparse` 工具来做类型检查。那 `Sparse` 又是什么？它是一种静态分析工具(static analysis tool)，用于在 Linux 内核源代码中发现各种类型的漏洞，一直都是比较神秘的角色，最初由 Linus Torvalds 写的，后来 Linus 没有继续维护，直到去年的冬天，它才有了新的维护者 Josh Triplett。

可能还会有第三个问题，`usb_alloc_urb` 也没做多少事，它做的那些咱们自己很容易就能做了，为什么还说驱动里一定要使用它来创建 `urb` 呢？按照 C++ 的说法，它就是 `urb` 的构造函数，

构造函数是创建对象的唯一方式，如果你说 C++ 里面使用位复制去复制一个简单对象给新对象就没使用构造函数，那是你不知道，C++ 的 ARM 里将这时的构造函数称为 **trivial copy constructor**。再说，现在它做的这些事情，以后还是做这些吗？它将创建 **urb** 的工作给包装了，我们只管调用就是了，这就是孙子兵法里有“以不变应万变”。

对应的，当然还会有一个析构函数，是销毁 **urb** 的，也在 **urb.c** 中定义。

```
81 void usb_free_urb(struct urb *urb)
82 {
83     if (urb)
84         kref_put(&urb->kref, urb_destroy);
85 }
```

**usb\_free\_urb** 更潇洒，只调用 **kref\_put** 将 **urb** 的引用计数减 1，减了之后如果变为 0，也就是没人再用它了，就调用 **urb\_destroy** 将它销毁掉。

接着看 **usb\_fill\_control\_urb** 函数，初始化刚才创建的控制 **urb**，你要想使用 **urb** 进行 USB 传输，不是光为它申请一点内存就够的，你得为它初始化，填充点实实在在的内容。它是在 **include/linux/usb.h** 中定义的内联函数。

```
1175 static inline void usb_fill_control_urb (struct urb *urb,
1176                                         struct usb_device *dev,
1177                                         unsigned int pipe,
1178                                         unsigned char *setup_packet,
1179                                         void *transfer_buffer,
1180                                         int buffer_length,
1181                                         usb_complete_t complete_fn,
1182                                         void *context)
1183 {
1184     spin_lock_init(&urb->lock);
1185     urb->dev = dev;
1186     urb->pipe = pipe;
1187     urb->setup_packet = setup_packet;
1188     urb->transfer_buffer = transfer_buffer;
1189     urb->transfer_buffer_length = buffer_length;
1190     urb->complete = complete_fn;
1191     urb->context = context;
1192 }
```

这个函数基本上都是赋值语句，把你在参数中指定的值充实给刚刚创建的 **urb**，**urb** 的元素有很多，这里只是填充了一部分，剩下那些元素不是控制传输管不着的，就是自有安排可以不用去管的。

你想想，一个 **struct urb** 结构要应付四种传输类型，每种传输类型总会有一点自己特别的要求，总会有些元素专属于某种传输类型，而其他传输类型不用管的。如果按 C++ 的做法，这称不上是一个好的设计思想，应该有一个基类 **urb**，里面放点儿四种传输类型公用的函数，比如 **pipe**，**transfer\_buffer** 等，再搞几个子类，如 **control\_urb**，**bulk\_urb** 等，专门应付具体的传输类型，如果不用什么虚函数，实际的时间和空间消耗也不会增加什么。但是实在没必要这么做，

现在内核的结构已经够多了，你创建什么类型的 `urb`，就填充相关的一些字段好了，况且写 USB Core 的人已经给咱们提供了不同传输类型的初始化函数，就像上面的 `usb_fill_control_urb` 函数，对于批量传输有 `usb_fill_bulk_urb` 函数，对于中断传输有 `usb_fill_int_urb` 函数，一般来说这也足够了，下面就查看 `usb_fill_control_urb` 函数的这俩孪生兄弟。

```
1207 static inline void usb_fill_bulk_urb (struct urb *urb,
1208                                     struct usb_device *dev,
1209                                     unsigned int pipe,
1210                                     void *transfer_buffer,
1211                                     int buffer_length,
1212                                     usb_complete_t complete_fn,
1213                                     void *context)
1214 {
1215     spin_lock_init(&urb->lock);
1216     urb->dev = dev;
1217     urb->pipe = pipe;
1218     urb->transfer_buffer = transfer_buffer;
1219     urb->transfer_buffer_length = buffer_length;
1220     urb->complete = complete_fn;
1221     urb->context = context;
1222 }
1223
1242 static inline void usb_fill_int_urb (struct urb *urb,
1243                                     struct usb_device *dev,
1244                                     unsigned int pipe,
1245                                     void *transfer_buffer,
1246                                     int buffer_length,
1247                                     usb_complete_t complete_fn,
1248                                     void *context,
1249                                     int interval)
1250 {
1251     spin_lock_init(&urb->lock);
1252     urb->dev = dev;
1253     urb->pipe = pipe;
1254     urb->transfer_buffer = transfer_buffer;
1255     urb->transfer_buffer_length = buffer_length;
1256     urb->complete = complete_fn;
1257     urb->context = context;
1258     if (dev->speed == USB_SPEED_HIGH)
1259         urb->interval = 1 << (interval - 1);
1260     else
1261         urb->interval = interval;
1262     urb->start_frame = -1;
1263 }
```

负责批量传输的 `usb_fill_bulk_urb` 函数和负责控制传输的 `usb_fill_control_urb` 函数相比，只是少初始化了一个 `setup_packet`，因为批量传输里没有 Setup 包的概念，中断传输里也没有，所以 `usb_fill_int_urb` 里也没有初始化 `setup_packet`。不过 `usb_fill_int_urb` 函数比那两个函数还是多了点儿内容的，因为它有一个 `interval`，比控制传输和批量传输多了一个表达自己期望的权利，1258 行还做了一次判断，如果是高速传输就怎么办，否则又该怎么办，主要是高速传输和低速传输/全速传输的间隔时间单位不一样，低速传输/全速传输的单位为帧，高速传输的单位为微帧，还要经过 2 的  $(bInterval-1)$  次方计算一下。至于 1262 行 `start_frame`，它是给等时传输用

的，这里自然就设置为-1，关于为什么既然 `start_frame` 是等时传输用的这里还要设置那么一下。

我们很快发现 `usb_fill_control_urb` 的孪生兄弟里，少了等时传输对应的初始化函数。对于等时传输，`urb` 里是可以指定进行多次传输的，你必须一个一个地对那个变长数组 `iso_frame_desc` 里的内容进行初始化。难道你能想出一个办法创建一个适用于各种情况等时传输的初始化函数？我是不能。如果想不出来，使用 `urb` 进行等时传输时，还是老老实实地对里面相关的字段一个一个地填内容吧。如果想找个例子参考一下别人是怎么初始化的，可以去找个摄像头的驱动程序，或者其他 USB 音视频设备的驱动程序看一看，内核中也有一些。

现在，你应该还记得是因为要设置设备的地址，让设备进入 `Address` 状态，调用了 `usb_control_msg`，才遇到 `usb_fill_control_urb` 的，参数中的 `setup_packet` 就是之前创建和赋好值的 `struct usb_ctrlrequest` 结构体，设备的地址已经在 `struct usb_ctrlrequest` 结构体 `wValue` 字段里了。这次控制传输并没有 `DATA transaction` 阶段，也并不需要 `urb` 提供什么 `transfer_buffer` 缓冲区，所以 `transfer_buffer` 应该传递一个 `NULL`，当然 `transfer_buffer_length` 也就为 0 了。有意思的是，这时候传递进来的结束处理函数 `usb_api_blocking_completion`，可以看一下当这次控制传输已经完成，设备地址已经设置好后，接着做了些什么，它的定义在 `drivers/usb/core/message.c` 里。

```
21 static void usb_api_blocking_completion(struct urb *urb)
22 {
23     complete((struct completion *)urb->context);
24 }
```

这个函数更简洁，就一句，没有前面说的释放 `urb`，也没有重新提交它。设置一个地址就可以了，没必要再将它提交给 `HCD`，你就是再提交多少次，设置多少次，也只能有一个地址，USB 的世界里不提倡囤积居奇。那在这里仅仅一句里面都做了些什么？接着往下看。

然后就是第三个基本点，`usb_start_wait_urb` 函数，将前面历经千辛万苦创建和初始化的 `urb` 提交给 USB core，让它移交给特定的主机控制器驱动进行处理，然后等待 `HCD` 回馈的结果，如果等待的时间超过了预期的限度，它不会再等。它在 `message.c` 中定义。

```
33 static int usb_start_wait_urb(struct urb *urb, int timeout, int
*actual_length)
34 {
35     struct completion done;
36     unsigned long expire;
37     int status;
38
39     init_completion(&done);
40     urb->context = &done;
41     urb->actual_length = 0;
42     status = usb_submit_urb(urb, GFP_NOIO);
43     if (unlikely(status))
44         goto out;
45
46     expire = timeout ? msecs_to_jiffies(timeout) : MAX_SCHEDULE_TIMEOUT;
47     if (!wait_for_completion_timeout(&done, expire)) {
```



```

48
49         dev_dbg(&urb->dev->dev,
50                 "%s timed out on ep%d%s len=%d/%d\n",
51                 current->comm,
52                 usb_pipeendpoint(urb->pipe),
53                 usb_pipein(urb->pipe) ? "in" : "out",
54                 urb->actual_length,
55                 urb->transfer_buffer_length);
56
57         usb_kill_urb(urb);
58         status = urb->status == -ENOENT ? -ETIMEDOUT : urb->status;
59     } else
60         status = urb->status;
61 out:
62     if (actual_length)
63         *actual_length = urb->actual_length;
64
65     usb_free_urb(urb);
66     return status;
67 }

```

35 行，定义了一个 `struct completion` 结构体。`completion` 是内核中一个比较简单的同步机制，一个线程可以通过它来通知另外一个线程某件事情已经做完了。

`completion` 机制也同样是这么回事儿，你的代码执行到某个地方，需要再忙其他的事情，就新开一个线程，让它去忙了，然后接着忙自己的，想知道那边儿的结果了，就停在某个地方等着，那边儿忙完了会通知一下已经有结果了，于是代码又可以继续往下走。

`completion` 机制围绕 `struct completion` 结构去实现，有两种使用方式，一种是通过 `DECLARE_COMPLETION` 宏在编译时就创建好 `struct completion` 的结构体，另外一种就是上面的使用形式，运行时才创建的。先在 35 行定义一个 `struct completion` 结构体，然后在 39 行使用 `init_completion` 去初始化。光是创建 `struct completion` 的结构体没用，关键的是如何通知任务已经完成了，和怎么去等候完成的好消息。我们的计算机在网上下载数据完成后可能会用声音、对话框等多种方式来通知你，同样这里用来通知已经完成任务了的函数也不止一个。

```

void complete(struct completion *c);
void complete_all(struct completion *c);

```

`complete` 只通知一个等候的线程，`complete_all` 可以通知所有等候的线程。

你不可能毫无限度地等下去，所以针对不同的情况，等候的方式就有好几种，都在 `kernel/sched.c` 中定义。

```

void wait_for_completion(struct completion *c);
unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout);
int wait_for_completion_interruptible(struct completion *x);
unsigned long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout);

```

47 行使用的就是 `wait_for_completion_timeout`，设定一个时间限度，然后在那里候着，直

到得到通知，或者超过时间。既然有等的一方，也总得有通知的一方吧，不然岂不是每次都超时？记得上面刚出现过的结束处理函数 `usb_api_blocking_completion` 吗？它里面唯一的一句 `complete((struct completion *)urb->context)` 就是用来通知这里的 47 行。有疑问的话看 40 行，将刚刚初始过的 `struct completion` 结构体 `done` 的地址赋值给了 `urb->context`，47 行等的就是这个 `done`。再看 42 行，`usb_submit_urb` 函数将这个控制 `urb` 提交给 USB Core，它是异步的，也就是说提交了之后不会等到传输完成了才返回。

`usb_start_wait_urb` 函数将 `urb` 提交给 USB Core 去处理后，就停在 47 行等候 USB Core 和 HCD 的处理结果，而这个 `urb` 代表的控制传输完成之后会调用结束处理函数 `usb_api_blocking_completion`，从而调用 `complete` 来通知 `usb_start_wait_urb` 不用再等了，传输已经完成了，当然还有一种可能是 `usb_start_wait_urb` 已经等待超过了时间限度仍然没有接到通知。不管是哪种情况，`usb_start_wait_urb` 都可以不用再等，继续往下走了。

42 行，提交 `urb`，并返回一个值表示是否提交成功了，显然，成功的可能性要远远大于失败的可能性，所以接下来的判断加上了 `unlikely`，如果真的失败，那也就没必要在 47 行等通知了，直接到后面去吧。

46 行，计算超时值。超时值在参数中不是已经给了吗，还要计算什么？没错，你是在参数中是指定了自己能够忍受的最大时间限度，不过那是以 `ms` 为单位的，不过在 Linux 里时间概念必须得加上一个 `jiffy`，因为函数 `wait_for_completion_timeout` 里的超时参数是必须以 `jiffy` 为单位的。

`jiffy`，意思是瞬间，短暂的时间跨度。短暂到什么程度？Linux 里它表示两次时钟中断的间隔，时钟中断是由定时器周期性产生的，关于这个周期，内核中用一个很形象的变量来描述，就是 `HZ`，它是一个体系结构相关的常数。内核中还提供了专门的计数器来记录从系统引导开始所度过的 `jiffy` 值，每次时钟中断发生时，这个计数器就增加 1。

既然你指定的时间和 `wait_for_completion_timeout` 需要的时间单位不一致，就需要转换一下，`msecs_to_jiffies` 函数可以完成这个工作，它将 `ms` 值转化为相应的 `jiffy` 值。在 46 行里，`MAX_SCHEDULE_TIMEOUT` 比较陌生，在 `include/linux/sched.h` 里它被定义为 `LONG_MAX`，最大的长整型值，在 `include/linux/kernel.h` 里看一看。

```

23 #define INT_MAX      ((int) (~0U>>1))
24 #define INT_MIN      (-INT_MAX - 1)
25 #define UINT_MAX     (~0U)
26 #define LONG_MAX     ((long) (~0UL>>1))
27 #define LONG_MIN     (-LONG_MAX - 1)
28 #define ULONG_MAX    (~0UL)
29 #define LLONG_MAX    ((long long) (~0ULL>>1))
30 #define LLONG_MIN    (-LLONG_MAX - 1)
31 #define ULLONG_MAX   (~0ULL)

```

各种整型数的最大值最小值都在这里了，‘~’是按位取反，‘UL’是无符号长整型，‘ULL’

是 64 位的无符号长整型，‘<<’左移运算就是直接把所有位往左边儿移若干位，‘>>’右移运算比较容易搞混，主要是怎么去补缺。在 C 里主要就是无符号整数和有符号整数的之间的冲突，在往右移之后，空出来的那些空缺，对于无符号整数得补 0，对于有符号的，得补上符号位。

还是讲一下 LONG\_MAX，上面定义为((long)(~0UL>>1))，0UL 按位取反之后全为 1 的无符号长整型，向右移 1 位，左边空出来的位置补 0，这个数对于无符号的 long 来说不算什么，但是再经过 long 这么强制转化一下变为有符号的长整型，它就是老大了。

现在你可以清楚地知道在 46 行指定的超时时间被转化为相应的 jiffy 值，或者直接被设定为最大的 long 值。

47 行，等待通知，我们需要知道的是怎么去判断等待的结果，也就是 wait\_for\_completion\_timeout 的返回值代表什么意思？一般来说，一个函数返回了 0 代表一切顺利，可是 wait\_for\_completion\_timeout 返回 0 恰恰表示坏消息，表示直到超过了忍耐的极限仍没有接到任何的回馈；而返回了一个大于 0 的值则表示接到通知了，不管是完成了还是出错了总归是告诉这边儿不用再等了，这个值具体的含义就是距你设定的时限提前了多少时间。为什么会这样？你去看一看 wait\_for\_completion\_timeout 的定义就知道了，它是通过 schedule\_timeout 来实现超时的，schedule\_timeout 的返回值就是这个意思。

现在就很明显了，如果超时了，就会打印一些调试信息提醒一下，然后调用 usb\_kill\_urb 函数终止这个 urb，再将返回值设定一下。如果收到了通知，直接设定返回值，就接着往下走。

62 行，actual\_length 是用来记录实际传输的数据长度的，是 usb\_control\_msg 需要的。不要说这个值 urb 里本来就有保存，接下来的 65 行就用 usb\_free\_urb 把这个 urb 给销毁了。actual\_length 是从上头儿传递过来的一个指针，这里要先判断一下 actual\_length 是不是空的。

---

## 25. 设备的生命线（六）

现在来讲一下 usb\_submit\_urb 函数，这是一个有几百行的函数。

```
220 int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
221 {
222     int                pipe, temp, max;
223     struct usb_device  *dev;
224     int                is_out;
225
226     if (!urb || urb->hcpriv || !urb->complete)
227         return -EINVAL;
228     if (!(dev = urb->dev) ||
229         (dev->state < USB_STATE_DEFAULT) ||
230         (!dev->bus) || (dev->devnum <= 0))
```

```

231         return -ENODEV;
232     if (dev->bus->controller->power.power_state.event != PM_EVENT_ON
233         || dev->state == USB_STATE_SUSPENDED)
234         return -EHOSTUNREACH;
235
236     urb->status = -EINPROGRESS;
237     urb->actual_length = 0;
238
239     /* Lots of sanity checks, so HCDs can rely on clean data
240      * and don't need to duplicate tests
241      */
242     pipe = urb->pipe;
243     temp = usb_pipetype(pipe);
244     is_out = usb_pipeout(pipe);
245
246     if (!usb_pipecontrol(pipe) && dev->state < USB_STATE_CONFIGURED)
247         return -ENODEV;
248
249     /* FIXME there should be a sharable lock protecting us against
250      * config/altsetting changes and disconnects, kicking in here.
251      * (here == before maxpacket, and eventually endpoint type,
252      * checks get made.)
253      */
254
255     max = usb_maxpacket(dev, pipe, is_out);
256     if (max <= 0) {
257         dev_dbg(&dev->dev,
258             "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
259             usb_pipeendpoint(pipe), is_out ? "out" : "in",
260             __FUNCTION__, max);
261         return -EMSGSIZE;
262     }
263
264     /* periodic transfers limit size per frame/uframe,
265      * but drivers only control those sizes for ISO.
266      * while we're checking, initialize return status.
267      */
268     if (temp == PIPE_ISOCHRONOUS) {
269         int n, len;
270
271         /* "high bandwidth" mode, 1-3 packets/uframe? */
272         if (dev->speed == USB_SPEED_HIGH) {
273             int mult = 1 + ((max >> 11) & 0x03);
274             max &= 0x07ff;
275             max *= mult;
276         }
277
278         if (urb->number_of_packets <= 0)
279             return -EINVAL;
280         for (n = 0; n < urb->number_of_packets; n++) {
281             len = urb->iso_frame_desc[n].length;
282             if (len < 0 || len > max)
283                 return -EMSGSIZE;
284             urb->iso_frame_desc[n].status = -EXDEV;
285             urb->iso_frame_desc[n].actual_length = 0;
286         }
287     }
288
289     /* the I/O buffer must be mapped/unmapped, except when length=0 */

```

```

290     if (urb->transfer_buffer_length < 0)
291         return -EMSGSIZE;
292
293 #ifdef DEBUG
294     /* stuff that drivers shouldn't do, but which shouldn't
295      * cause problems in HCDs if they get it wrong.
296      */
297     {
298         unsigned int    orig_flags = urb->transfer_flags;
299         unsigned int    allowed;
300
301         /* enforce simple/standard policy */
302         allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
303                  URB_NO_INTERRUPT);
304         switch (temp) {
305         case PIPE_BULK:
306             if (is_out)
307                 allowed |= URB_ZERO_PACKET;
308             /* FALLTHROUGH */
309         case PIPE_CONTROL:
310             allowed |= URB_NO_FSBR; /* only affects UHCI */
311             /* FALLTHROUGH */
312         default:
313             /* all non-iso endpoints */
314             if (!is_out)
315                 allowed |= URB_SHORT_NOT_OK;
316             break;
317         case PIPE_ISOCHRONOUS:
318             allowed |= URB_ISO_ASAP;
319             break;
320         }
321         urb->transfer_flags &= allowed;
322
323         /* fail if submitter gave bogus flags */
324         if (urb->transfer_flags != orig_flags) {
325             err("BOGUS urb flags, %x --> %x",
326                orig_flags, urb->transfer_flags);
327             return -EINVAL;
328         }
329 #endif
330     /*
331      * Force periodic transfer intervals to be legal values that are
332      * a power of two (so HCDs don't need to).
333      *
334      * FIXME want bus->{intr,iso}_sched_horizon values here. Each HC
335      * supports different values... this uses EHCI/UHCI defaults (and
336      * EHCI can use smaller non-default values).
337      */
338     switch (temp) {
339     case PIPE_ISOCHRONOUS:
340     case PIPE_INTERRUPT:
341         /* too small? */
342         if (urb->interval <= 0)
343             return -EINVAL;
344         /* too big? */
345         switch (dev->speed) {
346         case USB_SPEED_HIGH: /* units are microframes */
347             // NOTE usb handles 2^15
348             if (urb->interval > (1024 * 8))

```

```

349         urb->interval = 1024 * 8;
350         temp = 1024 * 8;
351         break;
352     case USB_SPEED_FULL: /* units are frames/ msec */
353     case USB_SPEED_LOW:
354         if (temp == PIPE_INTERRUPT) {
355             if (urb->interval > 255)
356                 return -EINVAL;
357             // NOTE ohci only handles up to 32
358             temp = 128;
359         } else {
360             if (urb->interval > 1024)
361                 urb->interval = 1024;
362             // NOTE usb and ohci handle up to 2^15
363             temp = 1024;
364         }
365         break;
366     default:
367         return -EINVAL;
368     }
369     /* power of two? */
370     while (temp > urb->interval)
371         temp >>= 1;
372     urb->interval = temp;
373 }
374
375 return usb_hcd_submit_urb(urb, mem_flags);
376 }

```

这个函数虽然很长，目标却很简单，就是对 `urb` 做些前期处理后扔给 HCD。

226 行，一些有关存在性的判断，这个函数在开始时就要履行一下常规的检验，`urb` 为空，都没有初始化是不可以提交给 Core 的，Core 很生气，后果很严重，`hcpriv` 本来说好了留给 HCD 用的，自己不能偷偷先用了，HCD 很生气，后果也会很严重，`complete`，每个 `urb` 结束了都必须得调用一次 `complete` 代表的函数。

228 行，226 行是对 `urb` 本身的检验，这里是对 `urb` 的目的地 USB 设备的检验。要想让设备回应，它起码得达到 Default 状态。

设备编号 `devnum` 的值肯定是不能为负的了，那为什么为 0 也不行呢？Token 包的地址域里有 7 位是表示设备地址的，也就是说总共可以有 128 个地址来分配给设备，但是其中 0 号地址是被保留作为默认地址用的，任何一个设备处于 Default 状态还没有进入 Address 状态时都需要通过这个默认地址来响应主机的请求，所以 0 号地址不能分配给任何一个设备，Hub 为设备选择一个地址时，只有选择到一个大于 0 的地址，设备的生命线才会继续，因此说这里的 `devnum` 的值是不可能也不应该为 0 的。

因为要设置设备的地址，让设备进入 Address 状态，所以针对 SET\_ADDRESS 请求再查看这个 `devnum`。主机向设备发送 SET\_ADDRESS 请求时，如果设备处于 Default 状态，就是它现在的状态，指定一个非 0 值时，设备将进入 Address 状态；指定 0 值时，设备仍然会处于 Default

状态。所以说这里的 `devnum` 也是不能为 0 的。如果设备已经处于 `Address` 状态，指定一个非 0 值时，设备仍然会处于 `Address` 状态，只是将使用新分配的地址，一个设备只能占用一个地址，它是分配的，如果指定了一个 0 值，则设备将离开 `Address` 状态退回到 `Default` 状态。

232 行，`power`，`power_state`，`event`，还有 `PM_EVENT_ON` 都是电源管理核心里的内容，这里的目的是判断设备所在的那条总线的主机控制器有没有挂起，然后再判断设备本身是不是处于 `Suspended` 状态。

236 行，常规检查都做完了，`Core` 和 `HCD` 已经认同了这个 `urb`，就将它的状态设置为 `-EINPROGRESS`，表示从现在开始 `urb` 的控制权就在 `Core` 和 `HCD` 手里了，在驱动那里看不到这个状态。

237 行，这时还没开始传输，实际传输的数据长度当然为 0 了，这里进行初始化，也是为了防止以后哪里出错返回了，在驱动里可以检查。

242 行，这几行获得管道的类型及方向。

246 行，在设备进入 `Configured` 状态之前，主机只能使用控制传输，通过默认管道，也就是管道 0 来和设备进行交流。

255 行，获得端点的 `wMaxPacketSize`，看一看 `include/linux/usb.h` 中定义的这个函数：

```
1458 static inline __u16
1459 usb_maxpacket(struct usb_device *udev, int pipe, int is_out)
1460 {
1461     struct usb_host_endpoint      *ep;
1462     unsigned                       epnum = usb_pipeendpoint(pipe);
1463
1464     if (is_out) {
1465         WARN_ON(usb_pipein(pipe));
1466         ep = udev->ep_out[epnum];
1467     } else {
1468         WARN_ON(usb_pipeout(pipe));
1469         ep = udev->ep_in[epnum];
1470     }
1471     if (!ep)
1472         return 0;
1473
1474     /* NOTE: only 0x07ff bits are for packet size... */
1475     return le16_to_cpu(ep->desc.wMaxPacketSize);
1476 }
```

这个函数是很简单的。要根据现有的信息获得一个端点的 `wMaxPacketSize` 当然是必须得获得该端点的描述符，我们知道每个 `struct usb_device` 结构体里都有两个数组：`ep_out` 和 `ep_in`，它们保存了各个端点对应的 `struct usb_host_endpoint` 结构体，只要知道该端点对应了这两个数组里的哪个元素就可以获得它的描述符了，这就需要知道该端点的端点号和方向，而端点的方向就是管道的方向，端点号也保存在 `pipe` 里。

你是不是会担心 `ep_out` 或 `ep_in` 数组都还空着，或者说没有保存对应的端点信息？不用担心它还是空的，即使是现在设备还刚从 `Powered` 走到 `Default`，连 `Address` 都没有，但是在使用 `usb_alloc_dev` 构造这个设备的 `struct usb_device` 时，就把它里面端点 0 的 `struct usb_host_endpoint` 结构体 `ep0` 指定给 `ep_out[0]` 和 `ep_in[0]` 了，而且后来还对 `ep0` 的 `wMaxPacketSize` 指定了值。不过如果真的没有从它们里面找到想要的端点的信息，那肯定就是哪里出错了，指定了错误的端点号，或其他什么原因，也就不再继续走下去了。

268 行，如果是等时传输就要进行一些特别的处理。272 行到 276 行这几行涉及高速、高带宽端点（`high speed, high bandwidth endpoint`）。前面提到 `interval` 时，说过每一帧或微帧最多只能有一次等时传输，完成一次等时 `transaction`，这么说主要是因为还没遇到高速高带宽的等时端点。高速高带宽等时端点每个微帧可以进行 2 次到 3 次等时 `transaction`，它和一般等时端点的主要区别也在这里，没必要专门为它设置个描述符类型，端点描述符 `wMaxPacketSize` 字段的 `bit 11~bit 12` 就是用来指定可以额外有几次等时 `transaction` 的，00 表示没有额外的 `transaction`，01 表示额外有 1 次，10 表示额外有 2 次，11 被保留。`wMaxPacketSize` 字段的前 10 位就是实际的端点每次能够处理的最大字节数。所以这几行意思就是如果是高速等时端点，获得它允许的额外等时 `transaction` 次数，和每次能够处理的最大字节数，再将它们相乘就得出了该等时端点每个微帧的所能传输的最大字节数。

278 行，`number_of_packets` 不大于 0 就表示这个等时 `urb` 没有指定任何一次等时传输，可以直接返回了。

280 行到 286 行，对等时 `urb` 里指定的各次等时传输分别进行处理。如果它们预期传输的数据长度比上面算出来的 `max` 值还要大，则返回。然后将它们实际传输的数据长度先置为 0，状态都先初始化为 `-EXDEV`，表示这次等时传输仅仅部分完成了，因为走到这里时传输都还没开始。

290 行，`transfer_buffer_length` 长度不能小于 0，等于 0 倒是可以的，毕竟不是什么时候都是有数据要传的。

293 行，见到 `#ifdef DEBUG` 我们都应该很高兴，这意味着一直到下面对应的 `#endif` 之间的代码都是调试时用的，对整体函数的作用无关痛痒。

338 行，`temp` 是上面计算出来的管道类型，那下面的各个 `case` 肯定是针对四种传输类型的了。不过可以发现，这里只“`case`”了等时传输和中断传输两种周期性的传输类型，因为是关于 `interval` 的处理，所以就没有控制传输和批量传输了。

342 行，这里保证等时和中断 `urb` 的 `interval` 值必须是大于 0 的，不然主机那边看不懂这是什么意思。

345 行，这里的 `switch` 根据目的设备的速度去“`case`”，设备速度有三种，`case` 也有三个。



前面已经说过，不同的速度，`urb->interval` 可以取不同的范围，不过你可能会发现那时说的最大值要比这里的限制要大一些，这是因为协议归协议，实现归实现。比如，对于 UHCI 来说，中断传输的 `interval` 值不能比 128 更大，而协议规定的最大值为 255。那么现在的问题是，`temp` 又是做什么用的？要注意 `urb->interval` 的单位是帧或者微帧，`temp` 只是为了调整它的值为 2 的次幂，这点从 370 行就可以看出来。

375 行，将 `urb` 交给 HCD，然后就进入 HCD 了。

本来 `usb_submit_urb` 函数到此应该结束了，但是它对于写驱动的来说太重要了，驱动里做的所有铺垫就是为了使用 `usb_submit_urb` 提交一个合适的 `urb` 给设备，满怀期待地等待着设备回馈需要的信息，然后才有接下来的处理，不然 USB 驱动只是一纸空谈毫无用处。

第一还是要再次强调一下，在调用 `usb_submit_urb` 提交你的 `urb` 之前，一定必须不得不要对它正确初始化，对于控制/中断/批量传输，Core 都提供了 `usb_fill_control_urb` 的几个孪生兄弟供你初始化使用，对于等时传输要自己手工一步一步小心翼翼地对 `urb` 的相关元素逐个赋值。`urb` 决定了整个 USB 驱动能否顺利运转。

第二，对于驱动来说，`usb_submit_urb` 是异步的，也就是说不用等传输完全完成就返回了，只要 `usb_submit_urb` 的返回值表示为 0，就表示已经提交成功了，`urb` 已经被 Core 和 HCD 认可了，接下来 Core 和 HCD 怎么处理就是它们的事了。

只要你提交成功了，不管是中间出了差错还是顺利完成，你指定的结束处理函数总是会调用，只有到这个时候，你才能够重新拿到 `urb` 的控制权，检查它是不是出错了，需要不需要释放或者是重新提交。第三，什么时候需要在结束处理函数中重新提交这个 `urb`？其实，我更想问的是对于中断/等时传输，是怎么实现让主机按一定周期去访问端点的？端点的描述符里已经指定了这个间隔时间，`urb` 里也有 `interval` 描述了这个间隔周期。可是 `urb` 一次只完成一次传输，即使等时传输也只完成有限次的传输，然后就在结束处理函数中返回了，`urb` 的控制权就完全属于驱动了，接下来的周期访问是怎么做到的？难道脱离 `urb` 后主机自己就去自动与端点通信了？即使是这样了，那通信的数据又在哪里，又怎么去得到这些数据？

事实上，第一次提交一个中断或等时的 `urb` 时，HCD 会根据 `interval` 判断一下自己是否能够满足你的需要，如果不能安排足够的带宽来完成这种周期性的传输，它是不可能批准请求的，如果它估量一下觉得可以满足，就会保留足够的带宽。但是这并不是就表明万事大吉了，HCD 是保留带宽了，可是驱动得保证在对应端点要处理的 `urb` 队列里总是有 `urb`，不能是空的，否则这个保留的带宽就会被“cancel”掉。

那么对于中断/等时传输，如何保证对应端点的 `urb` 队列里总是会有 `urb`？这就回到最开始的问题了。驱动需要在结束处理函数中重新初始化和提交刚刚完成的 `urb`，提醒一下，这个时候你是不能够修改 `interval` 的值，否则等待你的只能是错误信息。中断传输的例子可以去看一看触摸屏驱动，等时传输的例子可以去看一看摄像头驱动，看一看它们在结束处理函数中都做

了些什么，你就知道了。

第四，对于控制/批量/中断传输，实际上很多时候你可以不用创建 `urb`，不用对它初始化，不用调用 `usb_submit_urb` 来提交，USB Core 将这个过程分别封装在了 `usb_control_msg`、`usb_bulk_msg` 和 `usb_interrupt_msg` 这三个函数中，不同的是它们的实现是同步的，会去等待传输完全结束。咱们就是从 `usb_control_msg` 走过来的，所以这里只查看另外两个，它们都定义在 `drivers/usb/core/message.c` 里。

```
169 int usb_interrupt_msg(struct usb_device *usb_dev, unsigned int pipe,
170                       void *data, int len, int *actual_length, int timeout)
171 {
172     return usb_bulk_msg(usb_dev, pipe, data, len, actual_length, timeout);
173 }
```

`usb_interrupt_msg` 全部都借助 `usb_bulk_msg` 去完成了。

```
207 int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
208                  void *data, int len, int *actual_length, int timeout)
209 {
210     struct urb *urb;
211     struct usb_host_endpoint *ep;
212
213     ep = (usb_pipein(pipe) ? usb_dev->ep_in : usb_dev->ep_out)
214          [usb_pipeendpoint(pipe)];
215     if (!ep || len < 0)
216         return -EINVAL;
217
218     urb = usb_alloc_urb(0, GFP_KERNEL);
219     if (!urb)
220         return -ENOMEM;
221
222     if ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
223         USB_ENDPOINT_XFER_INT) {
224         pipe = (pipe & ~(3 << 30)) | (PIPE_INTERRUPT << 30);
225         usb_fill_int_urb(urb, usb_dev, pipe, data, len,
226                         usb_api_blocking_completion, NULL,
227                         ep->desc.bInterval);
228     } else
229         usb_fill_bulk_urb(urb, usb_dev, pipe, data, len,
230                           usb_api_blocking_completion, NULL);
231
232     return usb_start_wait_urb(urb, timeout, actual_length);
233 }
```

首先根据指定的 `pipe` 获得端点的方向和端点号，然后从设备 `struct usb_device` 结构体的 `ep_in` 或 `ep_out` 数组里得到端点对应的 `struct usb_host_endpoint` 结构体，接着调用 `usb_alloc_urb` 创建 `urb`。

因为这个函数可能是从 `usb_interrupt_msg` 调用过来的，所以接下来要根据端点描述符的 `bmAttributes` 字段获取传输的类型，判断究竟是中断传输还是批量传输，如果是中断传输的话还要修改 `pipe` 的类型，防止万一被其他函数直接调用 `usb_bulk_msg` 来完成中断传输。

不管是中断传输还是批量传输，都要调用 `usb_fill_xxx_urb` 函数来初始化，最后，和 `usb_control_msg` 一样，调用 `usb_start_wait_urb` 函数。

## 26. 设备的生命线（七）

在 HCD 这个片区域里，王中之王就是 `drivers/usb/core/hcd.h` 中定义的 `struct usb_hcd`。

```

58 struct usb_hcd {
59
60     /*
61      * housekeeping
62      */
63     struct usb_bus      self;          /* hcd is-a bus */
64     struct kref          kref;         /* reference counter */
65
66     const char          *product_desc; /* product/vendor string */
67     char                irq_descr[24]; /* driver + bus # */
68
69     struct timer_list    rh_timer;     /* drives root-hub polling */
70     struct urb           *status_urb;   /* the current status urb */
71 #ifdef CONFIG_PM
72     struct work_struct   wakeup_work;   /* for remote wakeup */
73 #endif
74
75     /*
76      * hardware info/state
77      */
78     const struct hc_driver *driver;     /* hw-specific hooks */
79
80     /* Flags that need to be manipulated atomically */
81     unsigned long        flags;
82 #define HCD_FLAG_HW_ACCESSIBLE 0x00000001
83 #define HCD_FLAG_SAW_IRQ      0x00000002
84
85     unsigned             rh_registered:1; /* is root hub registered? */
86
87     /* The next flag is a stopgap, to be removed when all the HCDs
88      * support the new root-hub polling mechanism. */
89     unsigned             uses_new_polling:1;
90     unsigned             poll_rh:1;     /* poll for rh status? */
91     unsigned             poll_pending:1; /* status has changed? */
92     unsigned             wireless:1;    /* Wireless USB HCD */
93
94     int                 irq;            /* irq allocated */
95     void __iomem         *regs;         /* device memory/io */
96     u64                 rsrc_start;     /* memory/io resource start */
97     u64                 rsrc_len;      /* memory/io resource length */
98     unsigned             power_budget;  /* in mA, 0 = no limit */
99
100 #define HCD_BUFFER_POOLS      4
101     struct dma_pool        *pool [HCD_BUFFER_POOLS];
102

```

```

103     int                                state;
104 #define __ACTIVE                        0x01
105 #define __SUSPEND                       0x04
106 #define __TRANSIENT                     0x80
107
108 #define HC_STATE_HALT                    0
109 #define HC_STATE_RUNNING                 (__ACTIVE)
110 #define HC_STATE_QUIESCING              (__SUSPEND|__TRANSIENT|__ACTIVE)
111 #define HC_STATE_RESUMING               (__SUSPEND|__TRANSIENT)
112 #define HC_STATE_SUSPENDED              (__SUSPEND)
113
114 #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
115 #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
116
117     /* more shared queuing code would be good; it should support
118     * smarter scheduling, handle transaction translators, etc;
119     * input size of periodic table to an interrupt scheduler.
120     * (ohci 32, uhci 1024, ehci 256/512/1024).
121     */
122
123     /* The HC driver's private data is stored at the end of
124     * this structure.
125     */
126     unsigned long hcd_priv[0]
127         __attribute__((aligned (sizeof(unsigned long))));
128 };

```

63 行，又一个结构体，`struct usb_bus` 里还有 `self`。

为什么这里会用这么一个戏剧性的词汇——`self`？我在前面提到过，一个主机控制器就会连出一条 USB 总线，主机控制器驱动用 `struct usb_hcd` 结构表示，一条总线用 `struct usb_bus` 结构表示。`struct usb_bus` 在 `include/linux/usb.h` 中定义。

```

276 struct usb_bus {
277     struct device *controller;    /* host/master side hardware */
278     int busnum;                  /* Bus number (in order of reg) */
279     char *bus_name;              /* stable id (PCI slot_name etc) */
280     u8 uses_dma;                 /* Does the host controller use DMA? */
281     u8 otg_port;                 /* 0, or number of OTG/HNP port */
282     unsigned is_b_host:1;        /* true during some HNP roleswitches */
283     unsigned b_hnp_enable:1;     /* OTG: did A-Host enable HNP? */
284
285     int devnum_next;             /* Next open device number in
286                                 * round-robin allocation */
287
288     struct usb_devmap devmap;    /* device address allocation map */
289     struct usb_device *root_hub; /* Root hub */
290     struct list_head bus_list;   /* list of busses */
291
292     int bandwidth_allocated;     /* on this bus: how much of the time
293                                 * reserved for periodic (intr/iso)
294                                 * requests is used, on average?
295                                 * Units: microseconds/frame.
296                                 * Limits: Full/low speed reserve 90%,
297                                 * while high speed reserves 80%.
298                                 */
299     int bandwidth_int_reqs;      /* number of Interrupt requests */

```

```

300     int bandwidth_isoc_reqs;          /* number of Isoc. requests */
301
302 #ifdef CONFIG_USB_DEVICEFS
303     struct dentry *usbfs_dentry; /* usbfs dentry entry for the bus */
304 #endif
305     struct class_device *class_dev; /* class device for this bus */
306
307 #if defined(CONFIG_USB_MON)
308     struct mon_bus *mon_bus;          /* non-null when associated */
309     int monitored;                    /* non-zero when monitored */
310 #endif
311 };

```

277 行，`controller`，`struct usb_hcd` 包含了一个 `usb_bus`，这里就回应了一个 `controller`。那现在在 `struct usb_hcd` 里的 `self` 和 `struct usb_bus` 里的 `controller` 这两个词儿，它们到底是什么关系？其实对于写代码的人来说一个主机控制器和一条总线差不多是一码事儿，不用分得那么清楚，可以简单地说明它们都是用来描述主机控制器的，那为什么又分成了两个结构？

USB 主机控制器是一个设备，而且更多的时候它还是一个 PCI 设备，那它就应该纳入这个设备模型范畴之内，`struct usb_hcd` 结构中就得嵌入类似 `struct device` 或 `struct pci_dev` 这样的一个结构体，但是你仔细看一看，能不能在它里面发现这么一个成员？不能。但是再看一看 `struct usb_bus` 里第一个成员就是一个 `struct device` 结构体。

再以 UHCI 为例说明一下，都在 `host` 目录下的 `uhci`-族文件中，首先它是一个 `pci` 设备，要使用 `pci_register_driver` 注册一个 `struct pci_driver` 结构体 `uhci_pci_driver`。`uhci_pci_driver` 里又有一个 `probe`，在这个 `probe` 里，它调用 `usb_create_hcd` 来创建一个 `usb_hcd`，并初始化里面的 `self`，还将这个 `self` 里的 `controller` 设定为描述主机控制器 `pci_dev` 里的 `struct device` 结构体，从而将 `usb_hcd`、`usb_bus` 和 `pci_dev`，甚至设备模型都连接起来了。

再接着看一下 `uhci`-文件中定义的函数。只看它们的参数，你会发现参数中不是 `struct usb_hcd` 就是 `struct uhci_hcd`，而且那些函数的前面几行常常会有 `hcd_to_uhci` 或者 `uhci_to_hcd` 这样的函数在 `struct usb_hcd` 和 `struct uhci_hcd` 之间转换。`struct uhci_hcd` 是什么？它是 `uhci` 自己私有的一个结构体，每个具体的主机控制器都有这么一个类似的结构体。顺便看一下 `hcd_to_uhci` 或者 `uhci_to_hcd` 的定义你就会明白，每个主机控制器的这个私有结构体都藏在 `struct usb_hcd` 结构最后的 `hcd_priv` 变长数组里。

对于具体的主机控制器驱动来说，它们的眼里只有 `struct usb_hcd`，`struct usb_hcd` 结构，至于主机控制器驱动，就如同 `struct usb_device` 或 `struct usb_interface` 对于 USB 驱动。没有 `usb_create_hcd` 去创建 `usb_hcd`，就不会有 `usb_bus` 的存在。

而对于 Linux 设备模型来说，`struct usb_bus` 无疑要更亲切一些。总之，你可以把 `struct usb_bus` 当做只是嵌入到 `struct usb_hcd` 里面的一个结构体，它将 `struct usb_hcd` 要完成的一部分工作进行了封装，因为要描述一个主机控制器太复杂太难，于是就开了 `struct usb_bus` 去专门面对设备模型、`sysfs` 等。这也就是在前面说 `struct usb_hcd` 才是“王中之王”的原因。

你知道 Greg 他们是怎么描述这种奇妙的关系吗？他们把这个叫做 HCD bus-glue layer，并致力于“flatten out it”。这个关系早先是比较混沌的，现在要清晰一些，以后只会更清晰，struct usb\_hcd 越来越走上台前，struct usb\_bus 越来越走向幕后。

278 行，busnum，总线编号，你的计算机里总可以有多个主机控制器吧，自然也就可以有多条 USB 总线了，既然可以有多条 USB 总线，就要编号方便确认了。有关总线编号，可以看一看定义在 drivers/usb/core/hcd.c 里的这几行。

```
88 /* used when allocating bus numbers */
89 #define USB_MAXBUS          64
90 struct usb_busmap {
91     unsigned long busmap [USB_MAXBUS / (8*sizeof (unsigned long))];
92 };
93 static struct usb_busmap busmap;
```

在讲 struct usb\_device 函数中的 devnum 时候，说到过一个 devicemap，这里又有一个 busmap。当时分析说 devicemap 一共有 128 位，同理可知，这里的 busmap 一共有 64 位，也就是说最多可以有 64 条 USB 总线。

279 行，bus\_name。bus，总线；name，名字；bus\_name 即总线的名字。什么样的名字？要知道大多数情况下主机控制器都是一个 PCI 设备，那么 bus\_name 应该就是用来在 PCI 总线上标识 USB 主机控制器的名字，PCI 总线使用标准的 PCI ID 来标识 PCI 设备，所以 bus\_name 里保存的应该就是主机控制器对应的 PCI ID。UHCI 等调用 usb\_create\_hcd 创建 usb\_hcd 时确实是将它们对应 PCI ID 赋给了 bus\_name。

现在简单介绍一下 PCI ID。PCI spec 允许单个系统可以最多有 256 条 PCI 总线，对我们来说当然是太多了，但是对于一些特殊的系统，它可能还觉得这满足不了要求，于是所有的 PCI 总线又被划分为 domain，每个 PCI domain 又可以最多拥有 256 条总线，而且每条总线上又可以支持 32 个设备。这些设备中还都可以是多功能板，它们还都可以最多支持 8 种功能。那系统怎么来区分每种功能的呢？总要知道它在哪个 domain，哪条总线，哪个设备板上吧。这么说还是太笼统了，你可以用 lspci 命令看一下。

```
00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge
(rev 01)
00:01.0 PCI bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev
01)
00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 08)
00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:07.2 USB Controller: Intel Corporation 82371AB/EB/MB PIIX4 USB
00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0f.0 VGA compatible controller: VMware Inc [VMware SVGA II] PCI Display Adapter
00:10.0 SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X
Fusion-MPT Dual Ultra320 SCSI (rev 01)
00:11.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE]
(rev 10)
00:12.0 Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev 02)
```

每行前面的数字就是所谓的“PCI ID”，每个 PCI ID 由 domain 号（16 位），总线编号（8 位），设备号（5 位），功能号（3 位）组成，不过这里 `lspci` 没有标明 domain 号，但对于一台普通计算机而言，一般也就只有一个 domain，0x0000。

280 行，`uses_dma`，表明这个主机控制器是否支持 DMA。主机控制器的一项重要工作就是在内存和 USB 总线之间传输数据，这个过程可以使用 DMA 或者不使用 DMA。不使用 DMA 的方式即所谓的 PIO 方式。DMA 代表着 Direct Memory Access，即直接内存访问，不需要 CPU 去干预。具体去看一看 PCI DMA 吧，因为一般来说主机控制器都是 PCI 设备，`uses_dma` 都在它们自己的 `probe` 函数中设置了。

281 行到 283 行，有关 `otg` 的代码。

285 行，`devnum_next`；288 行，`devmap`，早就说过 `devmap` 这张表了，`devnum_next` 中记录的就是这张表里下一个为 0 的位，里面为 1 的位都是已经被这条总线上的 USB 设备占据了的。

289 行，`root_hub`，Root Hub 在所有的 Hub 里面是那么特殊，还记得 USB 的那棵树吗？它就是那棵树的根，和 USB 主机控制器绑定在一起，其他的 hub 和设备都必须从它这儿延伸出去。正是因为这种特殊的关系，写代码的人就直接将它放在了 `struct usb_bus` 结构中。USB 主机控制器：USB 总线：Root Hub 为 1：1：1。

290 行，`bus_list`，在 `drivers/usb/core/hcd.c` 中定义有一个全局队列 `usb_bus_list`。

```
84 /* host controllers we manage */
85 LIST_HEAD (usb_bus_list);
86 EXPORT_SYMBOL_GPL (usb_bus_list);
```

它就是所有 USB 总线的组织。每次一条总线新添加进来，都要向这个组织靠拢，都要使用 `bus_list` 字段链接在这个队列上。

292 行，`bandwidth_allocated`，表明总线为中断传输和等时传输预留了多少带宽，协议中规定了，对于高速传输来说，最多可以有 80% 的带宽，对于低速传输和全速传输要多一点，可以达到 90% 的带宽。它的单位是 `ms`，表示一帧或微帧内有多少 `ms` 可以留给中断/等时传输用。

299 行，`bandwidth_int_reqs`；300 行，`bandwidth_isoc_reqs`，分别表示当前中断传输和等时传输的数量。

302 行到 304 行，是 `usbfs` 的，每条总线都对应于 `/proc/bus/usb` 下的一个目录。

305 行，`class_dev`，这里又牵涉设备模型中的一个概念——设备的 `class`，即设备的类。像前面提到的设备模型中的总线、设备、驱动三个核心概念，纯粹是从写驱动的角度看的，而这里的类则是面向于 Linux 的广大用户，它不管你是用什么接口，怎么去连接，它只管对用户来说提供了什么功能。一个 SCSI 硬盘和一个 ATA 硬盘对驱动来说是不相关的两个东西，但是对于用户来说，它们都是硬盘，都是用来备份文件。

设备模型与 sysfs 是分不开的，class 在 sysfs 里的体现就在 /sys/class 下面，可以去看一看。

```
atm          dma graphics hwmon          i2c_adapter input
mem          miscnet pci_bus             scsi_device scsi_disk
scsi_host    sound  spi_host spi_master  spi_transport tty
usb_device   usb_endpoint usb_host       vc          vtconsole
```

看到里面的 `usb_host` 了吧，它就是所有 USB 主机控制器的类，这些目录都是怎么来的呢？咱们还要追溯一下 USB 子系统的初始化函数 `usb_init`，它里面有下面这一段：

```
877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;
```

当时只是简单说这是用来初始化主机控制器的，在 `hcd.c` 里：

```
671 static struct class *usb_host_class;
672
673 int usb_host_init(void)
674 {
675     int retval = 0;
676
677     usb_host_class = class_create(THIS_MODULE, "usb_host");
678     if (IS_ERR(usb_host_class))
679         retval = PTR_ERR(usb_host_class);
680     return retval;
681 }
```

`usb_host_init` 所作的一切就是调用 `class_create` 创建了一个 `usb_host` 这样的类，你只要加载了 `usbcore` 模块就能在 `/sys/class` 下面看到有 `usb_host` 目录出现。既然 `usb_host` 目录表示 USB 主机控制器的类，那么它下面应该就对应各个具体的主机控制器了，你用 `ls` 命令就能看到 `usb_host1`、`usb_host2` 等这样的目录，它们每个都对应一个在你系统里实际存在的主机控制器，实际上在 `hcd.c` 里的 `usb_register_bus` 函数有以下这几行。

```
735 bus->class_dev = class_device_create(usb_host_class, NULL, MKDEV(0,0),
736                                     bus->controller, "usb_host%d", busnum);
```

这两行就是使用 `class_device_create` 在 `/sys/class/usb_host` 下面为每条总线创建了一个目录，目录名里的数字代表的就是每条总线的编号，`usb_register_bus` 函数是每个主机控制器驱动在 `probe` 里调用的，向 USB Core 注册一条总线，也可以说是注册一个主机控制器。

在 `struct usb_bus` 的最后，307 行到 310 行，`CONFIG_USB_MON` 是干什么用的？这要查看 `drivers/usb/mon` 目录下的 `Kconfig`。

```
5 config USB_MON
6     bool "USB Monitor"
7     depends on USB!=n
8     default y
9     help
10     If you say Y here, a component which captures the USB traffic
11     between peripheral-specific drivers and HC drivers will be built.
12     For more information, see <file:Documentation/usb/usbmon.txt>.
```



```

13
14         This is somewhat experimental at this time, but it should be safe.
15
16         If unsure, say Y.

```

文件中就这么多内容，从里面咱们可以知道，如果定义了 `CONFIG_USB_MON`，一个所谓的 USB Monitor，也就是 USB 监视器就会编进内核。这个 Monitor 是用来监视 USB 总线上的底层通信流的，相关的文件都在 `drivers/usb/mon` 下面。

## 27. 设备的生命线（八）

这个世界上不需要努力就能得到的东西只有一样，那就是年龄。所以要不怕苦不怕累，看完 `struct usb_bus` 函数，回到 `struct usb_hcd` 函数，继续努力地往下看。

64 行，`kref`，USB 主机控制器的引用计数。`struct usb_hcd` 也有自己专用的引用计数函数，在 `hcd.c` 文件中。

```

1526 static void hcd_release (struct kref *kref)
1527 {
1528     struct usb_hcd *hcd = container_of (kref, struct usb_hcd, kref);
1529
1530     kfree(hcd);
1531 }
1532
1533 struct usb_hcd *usb_get_hcd (struct usb_hcd *hcd)
1534 {
1535     if (hcd)
1536         kref_get (&hcd->kref);
1537     return hcd;
1538 }
1539 EXPORT_SYMBOL (usb_get_hcd);
1540
1541 void usb_put_hcd (struct usb_hcd *hcd)
1542 {
1543     if (hcd)
1544         kref_put (&hcd->kref, hcd_release);
1545 }
1546 EXPORT_SYMBOL (usb_put_hcd);

```

66 行，`product_desc`，主机控制器的产品描述字符串，对于 UHCI，它为“UHCI Host Controller”，对于 EHCI，它为“EHCI Host Controller”。

67 行，`irq_descr[24]`，这里面保存的是“ehci-hcd:usb1”之类的字符串，也就是驱动的名称再加上总线编号。

71 行到 73 行，电源管理。

78 行, driver, 每个主机控制器驱动都有一个 struct hc\_driver 结构体。查看它在 hcd.h 中的定义。

```

149 struct hc_driver {
150     const char      *description; /* "ehci-hcd" etc */
151     const char      *product_desc; /* product/vendor string */
152     size_t          hcd_priv_size; /* size of private data */
153
154     /* irq handler */
155     irqreturn_t      (*irq) (struct usb_hcd *hcd);
156
157     int              flags;
158 #define HCD_MEMORY      0x0001      /* HC regs use memory (else I/O) */
159 #define HCD_USB11      0x0010      /* USB 1.1 */
160 #define HCD_USB2      0x0020      /* USB 2.0 */
161
162     /* called to init HCD and root hub */
163     int              (*reset) (struct usb_hcd *hcd);
164     int              (*start) (struct usb_hcd *hcd);
165
166     /* NOTE: these suspend/resume calls relate to the HC as
167      * a whole, not just the root hub; they're for PCI bus glue.
168      */
169     /* called after suspending the hub, before entering D3 etc */
170     int              (*suspend) (struct usb_hcd *hcd, pm_message_t message);
171
172     /* called after entering D0 (etc), before resuming the hub */
173     int              (*resume) (struct usb_hcd *hcd);
174
175     /* cleanly make HCD stop writing memory and doing I/O */
176     void              (*stop) (struct usb_hcd *hcd);
177
178     /* shutdown HCD */
179     void              (*shutdown) (struct usb_hcd *hcd);
180
181     /* return current frame number */
182     int              (*get_frame_number) (struct usb_hcd *hcd);
183
184     /* manage i/o requests, device state */
185     int              (*urb_enqueue) (struct usb_hcd *hcd,
186                                     struct usb_host_endpoint *ep,
187                                     struct urb *urb,
188                                     gfp_t mem_flags);
189     int              (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
190
191     /* hw synch, freeing endpoint resources that urb_dequeue can't */
192     void              (*endpoint_disable) (struct usb_hcd *hcd,
193                                             struct usb_host_endpoint *ep);
194
195     /* root hub support */
196     int              (*hub_status_data) (struct usb_hcd *hcd, char *buf);
197     int              (*hub_control) (struct usb_hcd *hcd,
198                                     u16 typeReq, u16 wValue, u16 wIndex,
199                                     char *buf, u16 wLength);
200     int              (*bus_suspend) (struct usb_hcd *);
201     int              (*bus_resume) (struct usb_hcd *);
202     int              (*start_port_reset) (struct usb_hcd *, unsigned port_num);
203     void              (*hub_irq_enable) (struct usb_hcd *);

```

```
204             /* Needed only if port-change IRQs are level-triggered */
205 };
```

与 usb\_driver 和 pci\_driver 一样，所有的“xxx\_driver”都有一堆函数指针，具体的主机控制器驱动就靠它们，这里只说一下函数指针之外的东西。

description 直白点儿说就是驱动的大名，比如对于 UHCI，它是“uhci\_hcd”，对于 EHCI，它就是“ehci\_hcd”。product\_desc 和 struct usb\_hcd 里的是 一样。hcd\_priv\_size 还是有点儿意思的，每个主机控制器驱动都会有一个私有结构体，藏在 struct usb\_hcd 最后的那个变长数组里，这个“变”也是相对的，在创建 usb\_hcd 时也得知道它能变多长，不然谁知道要申请多少内存，这个长度就 hcd\_priv\_size。

81 行，flags，属于 HCD 的一些标志，可用值就在 82 行和 83 行。它们什么意思？书到用时方恨少，flags 到用时才可知。

69 行，70 行，85 行到 91 行，这几行都是专为 root hub 服务的。一个主机控制器对应一个 Root Hub，即使在嵌入式系统里，硬件上主机控制器没有集成 Root Hub，软件上也需要虚拟一个出来，也就是所谓的 Virtual Root Hub。

它位置是特殊的，但需要提供的功能和其他 Hub 是没有什么差别的，仅仅是在和主机控制器的软硬件接口上有一些特别的规定。Root Hub 再怎么特别也始终是一个 Hub，是一个 USB 设备，也不能脱离 USB 这个大家庭，也要向组织注册，也要有自己的设备生命线。

92 行，wireless，无线 USB。

94 行到 97 行这几行都是与主机控制器的“娘家”PCI 有关的，说到 PCI 就不得不说到如图 1.20.1 所示的那张著名的表。

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00	厂商 ID	设备 ID	命令寄存器	状态寄存器	版本修订 ID	类代号	高速缓存线	延迟定时器	头类型	BIST						
0x10	基地址 0			基地址 1			基地址 2			基地址 3						
0x20	基地址 4			基地址 5			CardBus CIS 指针			子系统厂商 ID	子系统设备 ID					
0x30	扩展 ROM 基地址			保留					中断线	中断引脚	Min. Gnt	Max. Lat				

图 1.20.1 PCI 配置寄存器

在这张表里中断号等很多有用的东西都在里面准备好了。94 行的 irq 就躲在上面表儿里的倒数第四个 Byte，HCD 可以直接拿来用，根本就不用再去申请。接下来的 regs, rsrc\_start, rsrc\_len 就与中间的 Base Address0~5 脱不开关系了，牵涉到所谓的 I/O 内存和 I/O 端口，下面简单说一下。

大家都知道 CPU 是众人瞩目的焦点，但是它也不可能一个人完成工作，计算机运转是一个集体项目，CPU 也需要跟各种外设配合交流，它需要访问外设里的寄存器或者内存。

CPU 的差别主要表现在空间的差别。一些 CPU 芯片有两个空间，即 I/O 空间和内存空间，提供有专门访问外设 I/O 端口的指令；而另外一些 CPU 只有一个空间，即内存空间。外设的 I/O 端口可以映射在 I/O 空间也可以映射到内存空间，CPU 通过访问这两个空间来访问外设，I/O 空间有 I/O 空间访问的接口，内存空间有内存空间访问的接口。当然一些外设不但有寄存器，还有内存，也就是 I/O 内存，比如 EHCI/OHCI，它们需要映射到内存空间。但是不管映射到哪个空间，访问 I/O 端口还是 I/O 内存，CPU 必须知道它们映射后的地址，不然没有办法配合交流。

在图 1.20.1 中，中间的那些基地址就是保存 PCI 设备中 I/O 内存或 I/O 端口的首尾位置还有长度。驱动使用时要首先把它们给读出来，如果要映射到 I/O 空间，则要根据读到的值向系统申请 I/O 端口资源，如果要映射到内存空间，除了要申请内存资源，还要使用 `ioremap` 等进行映射。

96 行的 `rsrc_start` 和 97 行的 `rsrc_len` 保存的就是从表里读出来的主机控制器的 I/O 端口或内存的首地址和长度，95 行的 `regs` 保存的是调用 `ioremap_nocache` 映射后的内存地址。

98 行，`power_budget`，能够提供的电流。

101 行，`*pool [HCD_BUFFER_POOLS]`，几个 dma 池。因为 `HCD_BUFFER_POOLS` 在 100 行定义为 4，所以这里就表示每个主机控制器可以有 4 个 dma 池。

我们知道主机控制器是可以进行 DMA 传输的，再回到前面讲的 `struct urb`，它有两个成员，`transfer_dma` 和 `setup_dma`，前面只是说可以使用 `usb_buffer_alloc` 分配好 DMA 缓冲区给它们，然后再告诉 HCD urb 已经有了，HCD 就可以不用再进行复杂的 DMA 映射了。但并没有提到这个获取的 DMA 缓冲区是从哪里来的。这里所说的 DMA 池子里来的了。

像创建或销毁线程、数据库连接时比较消耗资源，可以先建一个池子预先创建好一批线程放里面，用时就从里面取出来，不用时就再放里面。

当然，DMA 池还有其他的作用。一般来说 DMA 映射获得的都是以页为单位的内存，urb 不需要这么大的，如果需要比较小的 DMA 缓冲区，就离不开 DMA 池了。还是查看主机控制器的这几个池子在 `buffer.c` 文件中是怎么创建的。

```
52 int hcd_buffer_create(struct usb_hcd *hcd)
53 {
54     char          name[16];
55     int           i, size;
56
57     if (!hcd->self.controller->dma_mask)
58         return 0;
59
```

```

60     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
61         if (!(size = pool_max [i]))
62             continue;
63         snprintf(name, sizeof name, "buffer-%d", size);
64         hcd->pool[i] = dma_pool_create(name, hcd->self.controller,
65                                     size, size, 0);
66         if (!hcd->pool [i]) {
67             hcd_buffer_destroy(hcd);
68             return -ENOMEM;
69         }
70     }
71     return 0;
72 }

```

这里首先要判断一下这个主机控制器支持不支持 DAM，如果不支持的话再创建 DMA 池就是纯粹无稽之谈了。如果主机控制器支持 DMA，就逐个使用 `dma_pool_alloc` 来创建 DMA 池，如果创建失败了，就调用同一个文件中的 `hcd_buffer_destroy` 来将已经创建成功的池子给销毁掉。

```

82 void hcd_buffer_destroy(struct usb_hcd *hcd)
83 {
84     int i;
85
86     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
87         struct dma_pool *pool = hcd->pool[i];
88         if (pool) {
89             dma_pool_destroy(pool);
90             hcd->pool[i] = NULL;
91         }
92     }
93 }

```

这里调用的 `dma_pool_destroy` 和 `dma_pool_alloc`。每个 DMA 池子都要找到四个函数，一个用来创建，一个用来销毁，一个用来取内存，一个用来放内存。上面只遇到了创建和销毁的函数，还少两个取内存和放内存的函数，再去同一个文件中找一找。

```

100 void *hcd_buffer_alloc(
101     struct usb_bus *bus,
102     size_t size,
103     gfp_t mem_flags,
104     dma_addr_t *dma
105 )
106 {
107     struct usb_hcd *hcd = bus_to_hcd(bus);
108     int i;
109
110     /* some USB hosts just use PIO */
111     if (!bus->controller->dma_mask) {
112         *dma = ~(dma_addr_t) 0;
113         return kmalloc(size, mem_flags);
114     }
115
116     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
117         if (size <= pool_max [i])
118             return dma_pool_alloc(hcd->pool [i], mem_flags, dma);

```

```

119     }
120     return dma_alloc_coherent(hcd->self.controller, size, dma, 0);
121 }
122
123 void hcd_buffer_free(
124     struct usb_bus *bus,
125     size_t          size,
126     void            *addr,
127     dma_addr_t      dma
128 )
129 {
130     struct usb_hcd *hcd = bus_to_hcd(bus);
131     int            i;
132
133     if (!addr)
134         return;
135
136     if (!bus->controller->dma_mask) {
137         kfree(addr);
138         return;
139     }
140
141     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
142         if (size <= pool_max [i]) {
143             dma_pool_free(hcd->pool [i], addr, dma);
144             return;
145         }
146     }
147     dma_free_coherent(hcd->self.controller, size, addr, dma);
148 }

```

又可以找到两个函数，即 `dma_pool_alloc` 和 `dma_pool_free` 函数，现在可以凑齐四个函数了。不过不用管它们四个到底什么长相，只要它们能够干活儿就成了，这里要注意的是以下几个问题。

第一个问题是即使你的主机控制器不支持 DMA，这几个函数也是可以用的，只不过创建的不是 DMA 池子，存取的也不是 DMA 缓冲区。此时，DMA 池子不存在，`hcd_buffer_alloc` 获取的只是适用 `kmalloc` 申请的普通内存。当然，你必须在它没有利用价值时使用 `hcd_buffer_free` 将它释放掉。

第二个问题是 `size` 的问题，它们里面都有一个 `pool_max`，这是个同一个文件中定义的数组。

```

26 static const size_t    pool_max [HCD_BUFFER_POOLS] = {
27     /* platfor ms without dma-friendly caches might need to
28      * prevent cacheline sharing...
29      */
30     32,
31     128,
32     512,
33     PAGE_SIZE / 2
34     /* bigger --> allocate pages */
35 };

```

这个数组中定义的就是四个池子中每个池子里保存的 DMA 缓冲区的 `size`。注意这里虽说

只定义了四种 size，但是并不说明你使用 `hcd_buffer_alloc` 获取 DMA 缓冲区时不能指定更大的 size，如果这几个池子都满足不了要求，那就会使用 `dma_alloc_coherent` 建立一个新的 DMA 映射。还有，每个人的情况都不一样，不可能都会完全恰好和上面定义的四种种 size 一致，那也不用怕，使用这个 size 获取 DMA 缓冲区时，池子会选择一个略大一些的回馈过去。

还是回到 `struct usb_hcd` 中来。103 行，`state`，主机控制器的状态，紧挨着它的下面那些代码就是相关的可用值和宏定义。

## 28. 设备的生命线（九）

说完了 `struct usb_hcd` 和 `struct usb_bus` 函数，接下来就该查看 `usb_submit_urb()` 最后的那个遗留问题 `usb_hcd_submit_urb()` 函数了。

现在内核中有个很不好的现象，设计结构比较复杂，函数比较长。

```

921 int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
922 {
923     int                status;
924     struct usb_hcd      *hcd = bus_to_hcd(urb->dev->bus);
925     struct usb_host_endpoint *ep;
926     unsigned long       flags;
927
928     if (!hcd)
929         return -ENODEV;
930
931     usbmon_urb_submit(&hcd->self, urb);
932
933     /*
934      * Atomically queue the urb, first to our records, then to the HCD.
935      * Access to urb->status is controlled by urb->lock ... changes on
936      * i/o completion (normal or fault) or unlinking.
937      */
938
939     // FIXME: verify that quiescing hc works right (RH cleans up)
940
941     spin_lock_irqsave (&hcd_data_lock, flags);
942     ep = (usb_pipein(urb->pipe) ? urb->dev->ep_in : urb->dev->ep_out)
943         [usb_pipeendpoint(urb->pipe)];
944     if (unlikely (!ep))
945         status = -ENOENT;
946     else if (unlikely (urb->reject))
947         status = -EPERM;
948     else switch (hcd->state) {
949     case HC_STATE_RUNNING:
950     case HC_STATE_RESUMING:
951     do_it:
952         list_add_tail (&urb->urb_list, &ep->urb_list);
953         status = 0;

```

```

954         break;
955     case HC_STATE_SUSPENDED:
956         /* HC upstream links (register access, wakeup signaling) can work
957          * even when the downstream links (and DMA etc) are quiesced; let
958          * usbcore talk to the root hub.
959          */
960         if (hcd->self.controller->power.power_state.event==PM_EVENT_ON
961             && urb->dev->parent == NULL)
962             goto doit;
963         /* FALL THROUGH */
964     default:
965         status = -ESHUTDOWN;
966         break;
967     }
968     spin_unlock_irqrestore (&hcd_data_lock, flags);
969     if (status) {
970         INIT_LIST_HEAD (&urb->urb_list);
971         usbmon_urb_submit_error(&hcd->self, urb, status);
972         return status;
973     }
974
975     /* increment urb's reference count as part of giving it to the HCD
976      * (which now controls it). HCD guarantees that it either returns
977      * an error or calls giveback(), but not both.
978      */
979     urb = usb_get_urb (urb);
980     atomic_inc (&urb->use_count);
981
982     if (urb->dev == hcd->self.root_hub) {
983         /* NOTE: requirement on hub callers (usbfs and the hub
984          * driver, for now) that URBs' urb->transfer_buffer be
985          * valid and usb_buffer_{sync,unmap}() not be needed, since
986          * they could clobber root hub response data.
987          */
988         status = rh_urb_enqueue (hcd, urb);
989         goto done;
990     }
991
992     /* lower level hcd code should use *_dma exclusively,
993      * unless it uses pio or talks to another transport.
994      */
995     if (hcd->self.uses_dma) {
996         if (usb_pipecontrol (urb->pipe)
997             && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
998             urb->setup_dma = dma_map_single (
999                 hcd->self.controller,
1000                 urb->setup_packet,
1001                 sizeof (struct usb_ctrlrequest),
1002                 DMA_TO_DEVICE);
1003         if (urb->transfer_buffer_length != 0
1004             && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
1005             urb->transfer_dma = dma_map_single (
1006                 hcd->self.controller,
1007                 urb->transfer_buffer,
1008                 urb->transfer_buffer_length,
1009                 usb_pipein (urb->pipe)
1010                     ? DMA_FROM_DEVICE
1011                     : DMA_TO_DEVICE);
1012     }

```



```
1013
1014     status = hcd->driver->urb_enqueue (hcd, ep, urb, mem_flags);
1015 done:
1016     if (unlikely (status)) {
1017         urb_unlink (urb);
1018         atomic_dec (&urb->use_count);
1019         if (urb->reject)
1020             wake_up (&usb_kill_urb_queue);
1021         usbmon_urb_submit_error(&hcd->self, urb, status);
1022         usb_put_urb (urb);
1023     }
1024     return status;
1025 }
```

`usb_hcd_submit_urb` 函数是 `hcd.c` 里的，目标也很明确，就是将提交过来的 `urb` 指派给合适的主机控制器驱动程序。Core 目录下面以“`hcd`”打头的几个文件严格来说不能算是 HCD，只能算 HCDBI，即主机控制器驱动接口层，这用来衔接具体的主机控制器驱动和 USB Core 的。

924 行，`bus_to_hcd` 是用来获得 `struct usb_bus` 结构体对应的 `struct usb_hcd` 结构体，`urb` 要去的那个设备所在的总线是在设备生命线的开始就初始化好了的。`bus_to_hcd` 还有一个兄弟 `hcd_to_bus`，都在 `hcd.h` 中定义。

```
131 static inline struct usb_bus *hcd_to_bus (struct usb_hcd *hcd)
132 {
133     return &hcd->self;
134 }
135
136 static inline struct usb_hcd *bus_to_hcd (struct usb_bus *bus)
137 {
138     return container_of(bus, struct usb_hcd, self);
139 }
```

继续看 928 行，这是大多数函数开头儿必备的常规检验，如果 `usb_hcd` 都还是空的，那就返回吧。

931 行，`usbmon_urb_submit` 就是与前面 Greg 孕育出来的 USB Monitor 有关的，如果你编译内核时没有配置上 `CONFIG_USB_MON`，它就是一个空函数。

941 行，去获得一把锁，这把锁在 `hcd.c` 的开头儿就已经初始化好了，所以说是把全局锁。

```
102 /* used when updating hcd data */
103 static DEFINE_SPINLOCK(hcd_data_lock);
```

前面多次遇到过自旋锁，现在就简单介绍一下。它和信号量，还有前面提到的 `completion` 一样都是 Linux 里用来进行代码同步。为什么要进行同步？要知道在 Linux 这个庞大复杂的世界里，可能同时有多个线程，那么只要它们互相之间有一定的共享，就必须要保证某一个线程操作这个共享时让其他线程知道。

自旋锁身为同步机制的一种，自然也有它独特的本事，它可以用在中断上下文或者原子上下文使用。上下文就是代码运行的环境，Linux 的这个环境使用二分法可以分成两种，能休眠

的环境和不能休眠的环境。像信号量和 `completion` 就只能用在可以休眠的环境，而自旋锁就用在不能休眠的环境里。

而 `usb_submit_urb` 还有 `usb_hcd_submit_urb` 必须得在两种环境里都能够使用，所以使用的是自旋锁，想一想 `urb` 的结束处理函数，它就是不能休眠的，但它里面必须得能够重新提交 `urb`。

再说一说 `hcd_data_lock` 这把锁都是用来保护什么的，为什么要使用它？主机控制器的 `struct usb_hcd` 结构体在它的驱动里早就初始化好了，但同一时刻是可能有多个 `urb` 向同一个主机控制器申请进行传输，可能有多个地方都希望访问它里面的内容。比如 948 行的 `state` 元素，显然就要同步了，`hcd_data_lock` 这把锁就是专门用来保护主机控制器的这个结构体的。

942 行，遇到多次也说过多次了，知道了 `pipe`，就可以从 `struct usb_device` 里的两个数组 `ep_in` 和 `ep_out` 里拿出对应端点的 `struct usb_host_endpoint` 结构体。

944 行到 973 行，这些行都是做检验的。

944 行，显然都走到这一步了，目的端点为空的可能性太小了，所以加上了 `unlikely`。

946 行，前面还说过，`urb` 里的这个 `reject`，只有 `usb_kill_urb` 有特权修改它，如果走到这里发现它的值大于 0 了，那就说明哪里调用了 `usb_kill_urb` 要终止这次传输，所以还是返回吧，不过这种可能性比较小，没人无聊到那种地步，总是刚提交就终止，吊主机控制器胃口，所以仍然加上 `unlikely`。

948 行，如果上面那两个检验都通过了，现在就“case”一下主机控制器的状态，如果为 `HC_STATE_RUNNING` 或 `HC_STATE_RESUMING` 就说明主机控制器这边儿没问题，尽管将这个 `urb` 往端点的 `urb` 队列里塞好了，952 行就是完成这个工作的。如果主机控制器的状态为 `HC_STATE_SUSPENDED`，但它的上行链路能够工作，而且这个 `urb` 是送往 Root Hub 的，则将其塞到 Root Hub 的 `urb` 队列里。

然后判断上面几次检验的结果，如果一切正常，则继续往下走，否则就返回吧。

979 行，检验都通过了，可以放心地增加 `urb` 的引用计数了。

980 行，将 `urb` 的 `use_count` 也增加 1，表示 `urb` 已经被 HCD 接受了，正在被处理着。你如果在这两个引用计数的差别还有疑问，再看一看前面讲解 `struct urb` 时的内容。

982 行，判断这个 `urb` 是不是流向 Root Hub 的，如果是，它就走向了 Root Hub 的生命线。不过，毕竟你更关注的是 USB 设备，应该很少有机会直接和 Root Hub 交流什么。

995 行，如果这个主机控制器支持 DMA，可你却没有告诉它 `URB_NO_SETUP_DMA_MAP` 或 `URB_NO_TRANSFER_DMA_MAP` 这两个标志，它就会认为你在 `urb` 里没有提供 DMA 的缓冲区，就会调用 `dma_map_single` 将 `setup_packet` 或 `transfer_buffer` 映射为 DMA 缓冲区。

1014 行，终于可以将 urb 交给具体的主机控制器驱动程序了。

到现在，设备已经可以进入 Address 状态。该继续看设备的那条生命线了。

## 29. 设备的生命线（十）

跟着设备的生命线走到现在，我算是明白了，事物的发展都是越往后就越高级越复杂，再看一看表 1.29.1，比较一下和上次那张表出现时有什么变化。

表 1.29.1

State	USB_STATE_ADDRESS
Speed	taken
ep0	ep0.urb_list, 描述符长度/类型, wMaxPacketSize

接下来设备的目标当然就是 Configured 了。

要进入 Configured 状态，就得去配置设备，当然不能是盲目地去配置，要知道设备是可能有多个配置的，所以要有选择、有目的、有步骤、有计划地去配置，但先去获得设备的设备描述符，message.c 中的 usb\_get\_device\_descriptor()就是 core 里专门干这个工作的。

```
860 int usb_get_device_descriptor(struct usb_device *dev, unsigned int size)
861 {
862     struct usb_device_descriptor *desc;
863     int ret;
864
865     if (size > sizeof(*desc))
866         return -EINVAL;
867     desc = kmalloc(sizeof(*desc), GFP_NOIO);
868     if (!desc)
869         return -ENOMEM;
870
871     ret = usb_get_descriptor(dev, USB_DT_DEVICE, 0, desc, size);
872     if (ret >= 0)
873         memcpy(&dev->descriptor, desc, size);
874     kfree(desc);
875     return ret;
876 }
```

这个函数比较的精练，先是准备了一个 struct usb\_device\_descriptor 结构体，然后就用它去调用 message.c 里的 usb\_get\_descriptor()获得设备描述符，获得之后再把得到的描述符复制到设备 struct usb\_device 结构体的 descriptor 成员里。因此，这个函数成功与否的关键就在 usb\_get\_descriptor()。其实对于写驱动的来说，眼里是只有 usb\_get\_descriptor() 没有 usb\_get\_device\_descriptor()的，不管你想获得哪种描述符都是要通过 usb\_get\_descriptor()，而 usb\_get\_device\_descriptor()是专属内核用的接口。

```
618 int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned
char index, void *buf, int size)
619 {
620     int i;
621     int result;
622
623     me mset(buf,0,size);    // Make sure we parse really received data
624
625     for (i = 0; i < 3; ++i) {
626         /* retry on length 0 or stall; some devices are flakey */
627         result = usb_control_ msg(dev, usb_rcvctrlpipe(dev, 0),
628                                   USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
629                                   (type << 8) + index, 0, buf, size,
630                                   USB_CTRL_GET_TIMEOUT);
631         if (result == 0 || result == -EPIPE)
632             continue;
633         if (result > 1 && ((u8 *)buf)[1] != type) {
634             result = -EPROTO;
635             continue;
636         }
637         break;
638     }
639     return result;
640 }
```

参数 `type` 就是用来区分不同的描述符的，协议中说了，`GET_DESCRIPTOR` 请求主要就是适用于三种描述符，即设备描述符、配置描述符和字符串描述符。参数 `index` 是要获得的描述符的序号，如果希望得到的这种描述符设备中可以有多 个，你需要指定获得其中的哪个，比如配置描述符就可以有多 个，不过对于设备描述符来说，只有一个，所以这里的 `index` 应该为 0。参数 `buf` 和 `size` 就是描述你用来放置获得的描述符的缓冲区的。

这个函数的内容挺单调的，主要就是调用了一个 `usb_control_ msg()`。这里要说的第一个问题是它的一堆参数，这就需要认真了解一下如图 1.29.1 所示的这张表。

bmRequestType	bRequest	wValue	wLength	Data
10000000B	GET_DESCRIPTOR	描述符的类型 和序号 (index)	描述符长度	描述符

图 1.29.1 GET\_DESCRIPTOR 请求

`GET_DESCRIPTOR` 请求的数据传输方向是 `device-to-host`，而且还是协议中规定所有设备都要支持的标准请求，也不是针对端点或者接口，而是针对设备的。所以 `bRequestType` 只能为 `0x80`，就是上面表里的 `10000000B`，也等于 628 行的 `USB_DIR_IN`。`wValue` 的高位字节表示描述符的类型，低位字节表示描述符的序号，所以就有 629 行的 `(type << 8) + index`。`wIndex` 对于字符串描述符应该设置为使用语言的 ID，对于其他的描述符应该设置为 0，所以也有了 629 行中间的那个 0。至于 `wLength`，就是描述符的长度，对于设备描述符，一般来说你都会指定为

USB\_DT\_DEVICE\_SIZE。

USB\_CTRL\_GET\_TIMEOUT 是定义在 include/linux/usb.h 里的一个宏，值为 5000，表示有 5s 的超时时间。

```
1330 #define USB_CTRL_GET_TIMEOUT    5000
1331 #define USB_CTRL_SET_TIMEOUT    5000
```

第二个问题就是为什么会有 3 次循环。这个又要归咎于一些不守规矩的生产设备厂商了，比如一些 USB 读卡器，一次请求还不一定能成功，但是设备描述符拿不到接下来就没法子走了，所以这里多试几次。至于 631 行到 636 行之间的代码都是判断是不是成功得到请求的描述符，这个版本的内核在这里的判断还比较混乱，就不多说了，你只要知道((u8 \*)buf)[1] != type 是用来判断获得描述符是不是请求的类型就可以了。

现在设备描述符已经有了，但是只有设备描述符是远远不够的，你从设备描述符里只能知道它一共支持几个配置，但是要配置一个设备时要知道具体每个配置有什么用，所以接下来就要获得各个配置的配置描述符，并且拿结果去充实 struct usb\_device 的 config、rawdescriptors 等相关元素。

core 内部并不直接调用上面的 usb\_get\_descriptor()去完成这个任务，而是调用 config.c 里的 usb\_get\_configuration()。

```
476 int usb_get_configuration(struct usb_device *dev)
477 {
478     struct device *ddev = &dev->dev;
479     int ncfg = dev->descriptor.bNumConfigurations;
480     int result = -ENOMEM;
481     unsigned int cfgno, length;
482     unsigned char *buffer;
483     unsigned char *bigbuffer;
484     struct usb_config_descriptor *desc;
485
486     if (ncfg > USB_MAXCONFIG) {
487         dev_warn(ddev, "too many configurations: %d, "
488             "using maximum allowed: %d\n", ncfg, USB_MAXCONFIG);
489         dev->descriptor.bNumConfigurations = ncfg = USB_MAXCONFIG;
490     }
491
492     if (ncfg < 1) {
493         dev_err(ddev, "no configurations\n");
494         return -EINVAL;
495     }
496
497     length = ncfg * sizeof(struct usb_host_config);
498     dev->config = kzalloc(length, GFP_KERNEL);
499     if (!dev->config)
500         goto err2;
501
502     length = ncfg * sizeof(char *);
503     dev->rawdescriptors = kzalloc(length, GFP_KERNEL);
504     if (!dev->rawdescriptors)
```

```

505         goto err2;
506
507     buffer = kmalloc(USB_DT_CONFIG_SIZE, GFP_KERNEL);
508     if (!buffer)
509         goto err2;
510     desc = (struct usb_config_descriptor *)buffer;
511
512     for (cfgno = 0; cfgno < ncfg; cfgno++) {
513         /* We grab just the first descriptor so we know how long
514          * the whole configuration is */
515         result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
516                                     buffer, USB_DT_CONFIG_SIZE);
517         if (result < 0) {
518             dev_err(ddev, "unable to read config index %d "
519                     "descriptor/%s\n", cfgno, "start");
520             dev_err(ddev, "chopping to %d config(s)\n", cfgno);
521             dev->descriptor.bNumConfigurations = cfgno;
522             break;
523         } else if (result < 4) {
524             dev_err(ddev, "config index %d descriptor too short "
525                     "(expected %i, got %i)\n", cfgno,
526                     USB_DT_CONFIG_SIZE, result);
527             result = -EINVAL;
528             goto err;
529         }
530         length = max((int) le16_to_cpu(desc->wTotalLength),
531                     USB_DT_CONFIG_SIZE);
532
533         /* Now that we know the length, get the whole thing */
534         bigbuffer = kmalloc(length, GFP_KERNEL);
535         if (!bigbuffer) {
536             result = -ENOMEM;
537             goto err;
538         }
539         result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
540                                     bigbuffer, length);
541         if (result < 0) {
542             dev_err(ddev, "unable to read config index %d "
543                     "descriptor/%s\n", cfgno, "all");
544             kfree(bigbuffer);
545             goto err;
546         }
547         if (result < length) {
548             dev_warn(ddev, "config index %d descriptor too short "
549                     "(expected %i, got %i)\n", cfgno, length, result);
550             length = result;
551         }
552
553         dev->rawdescriptors[cfgno] = bigbuffer;
554
555         result = usb_parse_configuration(&dev->dev, cfgno,
556                                     &dev->config[cfgno], bigbuffer, length);
557         if (result < 0) {
558             ++cfgno;
559             goto err;
560         }
561     }
562     result = 0;
563

```

```

564 err:
565     kfree(buffer);
566     dev->descriptor.bNumConfigurations = cfgno;
567 err2:
568     if (result == -ENOMEM)
569         dev_err(ddev, "out of memory\n");
570     return result;
571 }

```

要想得到配置描述符，最终都不可避免要向设备发送 GET\_DESCRIPTOR 请求，这就需要以 USB\_DT\_CONFIG 为参数调用 usb\_get\_descriptor 函数，也就需要知道该为获得的描述符准备多大的一个缓冲区。本来这个长度应该很明确地为 USB\_DT\_CONFIG\_SIZE，它表示的就是配置描述符的大小，但是实际上不是这么回事儿，USB\_DT\_CONFIG\_SIZE 只表示配置描述符本身的大小，并不表示 GET\_DESCRIPTOR 请求返回结果的大小。因为向设备发送 GET\_DESCRIPTOR 请求时，设备并不只是返回一个配置描述符，而是将这个配置下面的所有接口描述符、端点描述还有 class-或 vendor-specific 描述符都返回了。

那么这个总长度如何得到呢？在神秘的配置描述符里有一个神秘的字段 wTotalLength，它里面记录的就是这个总长度。那么问题就简单了，可以首先发送 USB\_DT\_CONFIG\_SIZE 个字节的请求过去，获得这个配置描述符的内容，从而获得那个总长度，然后以这个长度再请求一次，这样就可以获得一个配置下面所有的描述符内容了。上面的 usb\_get\_configuration() 采用的就是这个处理方法。

479 行，获得设备理配置描述符的数目。

486 行，USB\_MAXCONFIG 是 config.c 中定义的。

```

14 #define USB_MAXCONFIG 8 /* Arbitrary limit */

```

限制了一个设备最多只能支持 8 种配置拥有 8 个配置描述符，如果超出了这个限制，489 行就强制它为这个最大值。不过如果设备中没有任何一个配置描述符，什么配置都没有，那是不可能的，492 行这关就过不去。

498 行，struct usb\_device 里的 config 表示设备拥有的所有配置，你设备有多少个配置就为它准备多大的空间。

503 行，rawdescriptors，这是一个字符指针数组里的每一项都指向一个使用 GET\_DESCRIPTOR 请求去获取配置描述符时所得到的结果。

507 行，准备一个大小为 USB\_DT\_CONFIG\_SIZE 的缓冲区，第一次发送 GET\_DESCRIPTOR 请求要用到。

512 行，剩下的主要就是这个 for 循环了，获取每一个配置的那些描述符。

515 行，如上面所说的，首先发送 USB\_DT\_CONFIG\_SIZE 个字节请求，获得配置描述符

的内容。然后对返回的结果进行检验，知道为什么 523 行会判断结果是不是小于 4 吗？答案就在配置描述符中，里面的第 3 字节和第 4 字节就是 `wTotalLength`，只要得到前 4 个字节，就已经完成任务能够获得总长度了。

534 行，既然总长度已经有了，那么这里就为接下来的 `GET_DESCRIPTOR` 请求准备一个大点的缓冲区。

539 行，现在可以获得这个配置相关的所有描述符了。然后是对返回结果的检验，将得到的那一堆数据的地址赋给 `rawdescriptors` 数组里的指针。

555 行，从这个颇有韵味的数字 555 开始，你将会遇到另一个函数，它将对前面 `GET_DESCRIPTOR` 请求获得的那堆数据进行处理。

## 30. 设备的生命线（十一）

现在已经使用 `GET_DESCRIPTOR` 请求获取到了包含一个配置里所有相关描述符内容的一堆数据，这些数据是 `raw`，即原始的，所有数据不管是配置描述符、接口描述符还是端点描述符都挤在一起，所以得想办法将它们给分开，于是用到了 `usb_parse_configuration()` 函数。

```

264 static int usb_parse_configuration(struct device *ddev, int cfgidx,
265     struct usb_host_config *config, unsigned char *buffer, int size)
266 {
267     unsigned char *buffer0 = buffer;
268     int cfgno;
269     int nintf, nintf_orig;
270     int i, j, n;
271     struct usb_interface_cache *intfc;
272     unsigned char *buffer2;
273     int size2;
274     struct usb_descriptor_header *header;
275     int len, retval;
276     u8 inu ms[USB_MAXINTERFACES], nalts[USB_MAXINTERFACES];
277
278     memcpy(&config->desc, buffer, USB_DT_CONFIG_SIZE);
279     if (config->desc.bDescriptorType != USB_DT_CONFIG ||
280         config->desc.bLength < USB_DT_CONFIG_SIZE) {
281         dev_err(ddev, "invalid descriptor for config index %d: "
282             "type = 0x%X, length = %d\n", cfgidx,
283             config->desc.bDescriptorType, config->desc.bLength);
284         return -EINVAL;
285     }
286     cfgno = config->desc.bConfigurationValue;
287
288     buffer += config->desc.bLength;
289     size -= config->desc.bLength;
290
291     nintf = nintf_orig = config->desc.bNumInterfaces;

```



```

292     if (nintf > USB_MAXINTERFACES) {
293         dev_warn(ddev, "config %d has too many interfaces: %d, "
294                 "using maximum allowed: %d\n",
295                 cfgno, nintf, USB_MAXINTERFACES);
296         nintf = USB_MAXINTERFACES;
297     }
298
299     /* Go through the descriptors, checking their length and counting the
300      * number of altsettings for each interface */
301     n = 0;
302     for ((buffer2 = buffer, size2 = size);
303          size2 > 0;
304          (buffer2 += header->bLength, size2 -= header->bLength)) {
305
306         if (size2 < sizeof(struct usb_descriptor_header)) {
307             dev_warn(ddev, "config %d descriptor has %d excess "
308                     "Byte%s, ignoring\n",
309                     cfgno, size2, plural(size2));
310             break;
311         }
312
313         header = (struct usb_descriptor_header *) buffer2;
314         if ((header->bLength > size2) || (header->bLength < 2)) {
315             dev_warn(ddev, "config %d has an invalid descriptor "
316                     "of length %d, skipping remainder of the config\n",
317                     cfgno, header->bLength);
318             break;
319         }
320
321         if (header->bDescriptorType == USB_DT_INTERFACE) {
322             struct usb_interface_descriptor *d;
323             int inum;
324
325             d = (struct usb_interface_descriptor *) header;
326             if (d->bLength < USB_DT_INTERFACE_SIZE) {
327                 dev_warn(ddev, "config %d has an invalid "
328                         "interface descriptor of length %d, "
329                         "skipping\n", cfgno, d->bLength);
330                 continue;
331             }
332
333             inum = d->bInterfaceNumber;
334             if (inum >= nintf_orig)
335                 dev_warn(ddev, "config %d has an invalid "
336                         "interface number: %d but max is %d\n",
337                         cfgno, inum, nintf_orig - 1);
338
339             /* Have we already encountered this interface?
340              * Count its altsettings */
341             for (i = 0; i < n; ++i) {
342                 if (inu_ms[i] == inum)
343                     break;
344             }
345             if (i < n) {
346                 if (nalts[i] < 255)
347                     ++nalts[i];
348             } else if (n < USB_MAXINTERFACES) {
349                 inu_ms[n] = inum;
350                 nalts[n] = 1;

```

```

351         ++n;
352     }
353
354     } else if (header->bDescriptorType == USB_DT_DEVICE ||
355               header->bDescriptorType == USB_DT_CONFIG)
356         dev_warn(ddev, "config %d contains an unexpected "
357                  "descriptor of type 0x%X, skipping\n",
358                  cfgno, header->bDescriptorType);
359
360     } /* for ((buffer2 = buffer, size2 = size); ...) */
361     size = buffer2 - buffer;
362     config->desc.wTotalLength = cpu_to_le16(buffer2 - buffer0);
363
364     if (n != nintf)
365         dev_warn(ddev, "config %d has %d interface%s, different from "
366                  "the descriptor's value: %d\n",
367                  cfgno, n, plural(n), nintf_orig);
368     else if (n == 0)
369         dev_warn(ddev, "config %d has no interfaces?\n", cfgno);
370     config->desc.bNumInterfaces = nintf = n;
371
372     /* Check for missing interface numbers */
373     for (i = 0; i < nintf; ++i) {
374         for (j = 0; j < nintf; ++j) {
375             if (inu ms[j] == i)
376                 break;
377         }
378         if (j >= nintf)
379             dev_warn(ddev, "config %d has no interface number "
380                      "%d\n", cfgno, i);
381     }
382
383     /* Allocate the usb_interface_caches and altsetting arrays */
384     for (i = 0; i < nintf; ++i) {
385         j = nalts[i];
386         if (j > USB_MAXALTSETTING) {
387             dev_warn(ddev, "too many alternate settings for "
388                      "config %d interface %d: %d, "
389                      "using maximum allowed: %d\n",
390                      cfgno, inu ms[i], j, USB_MAXALTSETTING);
391             nalts[i] = j = USB_MAXALTSETTING;
392         }
393
394         len = sizeof(*intfc) + sizeof(struct usb_host_interface) * j;
395         config->intf_cache[i] = intfc = kzalloc(len, GFP_KERNEL);
396         if (!intfc)
397             return -ENOMEM;
398         kref_init(&intfc->ref);
399     }
400
401     /* Skip over any Class Specific or Vendor Specific descriptors;
402      * find the first interface descriptor */
403     config->extra = buffer;
404     i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
405                             USB_DT_INTERFACE, &n);
406     config->extralen = i;
407     if (n > 0)
408         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
409                n, plural(n), "configuration");

```

```

410     buffer += i;
411     size -= i;
412
413     /* Parse all the interface/altsetting descriptors */
414     while (size > 0) {
415         retval = usb_parse_interface(ddev, cfgno, config,
416                                     buffer, size, inu ms, nalts);
417         if (retval < 0)
418             return retval;
419
420         buffer += retval;
421         size -= retval;
422     }
423
424     /* Check for missing altsettings */
425     for (i = 0; i < nintf; ++i) {
426         intf = config->intf_cache[i];
427         for (j = 0; j < intf->num_altsetting; ++j) {
428             for (n = 0; n < intf->num_altsetting; ++n) {
429                 if (intf->altsetting[n].desc.
430                     bAlternateSetting == j)
431                     break;
432             }
433             if (n >= intf->num_altsetting)
434                 dev_warn(ddev, "config %d interface %d has no "
435                         "altsetting %d\n", cfgno, inu ms[i], j);
436         }
437     }
438
439     return 0;
440 }

```

其实前面也说到过的，使用 `GET_DESCRIPTOR` 请求时，得到的数据并不是杂乱无序的，而是有规可循的。一般来说，配置描述符后面跟的是第一个接口的接口描述符，接着是这个接口里第一个端点的端点描述符，如果有 `class`-和 `vendor-specific` 描述符的话，会紧跟在对应的标准描述符后面，不管接口有多少端点都是按照这个规律顺序排列。

当然有些设备生产厂商会特立独行一些，非要先返回第二个接口然后再返回第一个接口，但配置描述符后面总归先是接口描述符然后是端点描述符。

267 行，`buffer` 里保存的就是 `GET_DESCRIPTOR` 请求获得的数据，要解析这些数据，不可避免地要对 `buffer` 指针进行操作，这里先将它备份一下。

278 行，`config` 是参数中传递过来的，是设备 `struct usb_device` 结构体中的 `struct usb_host_config` 结构体数组 `config` 中的一员。不出意外的话，`buffer` 的前 `USB_DT_CONFIG_SIZE` 个字节对应的就是配置描述符，那么这里的意思就很明显了。

然后检验一下，查看这 `USB_DT_CONFIG_SIZE` 字节的内容究竟是不是正如我们所期待的，那样是一个配置描述符，如果不是，那 `buffer` 里的数据问题可就大了，没什么利用价值了，还是返回吧，不必要再接着解析了。

288 行, `buffer` 的前 `USB_DT_CONFIG_SIZE` 个字节已经理清了, 接下来该解析剩下的数据了, `buffer` 需要紧跟形势的发展, 位置和长度都要做相应的修正。

291 行, 获得这个配置所拥有的接口数目, 不能简单赋值就行了, 得知道系统里对这个数目是有 `USB_MAXINTERFACES` 这样的限制。如果数目比这个限制还大, 就改为 `USB_MAXINTERFACES`。

302 行到 360 行, 这函数就是统计记录一下这个配置里每个接口所拥有的设置数目。所以这里会提醒你一下, 千万别被迷惑了, 这个循环里使用的是 `buffer2` 和 `size2`, `buffer` 和 `size` 的两个替身。

306 行, 这里遇到一个新的结构 `struct usb_descriptor_header`, 在 `include/linux/usb/ch9.h` 中定义。

```
195 struct usb_descriptor_header {
196     __u8  bLength;
197     __u8  bDescriptorType;
198 } __attribute__((packed));
```

这个结构就包括了两个成员, 它们的前两个字节都是一样的, 一个表示描述符的长度, 一个表示描述符的类型。

那么为什么要专门放这么一个结构? 试想一下, 有一块数据缓冲区, 让你判断一下里面保存的是哪个描述符, 或者是其他什么东西, 你怎么做? 当然可以直接将它的前两个字节内容读出来, 判断 `bDescriptorType`, 再判断 `bLength`。不过这样的代码就好像你自己画的一副抽象画, 太艺术化了, 过了若干年连自己都不知道什么意思, 更别说别人了。313 行, 把 `buffer2` 指针转化为 `struct usb_descriptor_header` 的结构体指针, 然后就可以使用 ‘->’ 来取出 `bLength` 和 `bDescriptorType`。

那么 306 行就表示如果 `GET_DESCRIPTOR` 请求返回的数据里除了包括一个配置描述符外, 连两个字节都没有, 能看不能用。

321 行, 如果这是个接口描述符就说明这个配置的某个接口拥有一个设置是没有什么所谓的设置描述符的, 一个接口描述符就代表了存在一个设置, 接口描述里的 `bInterfaceNumber` 会指出这个设置隶属于哪个接口。那么这里除了是接口描述符还有可能是 `class`-和 `vendor-specific` 描述符。

325 行, 既然觉得这是一个接口描述符, 就把这个指针转化为 `struct usb_interface_descriptor` 结构体指针, 你可别被 C 语言里的这种指针游戏给转晕了, 一个地址如果代码不给它赋予什么意义, 它除了表示一个地址外就什么都不是。同样一个地址, 上面转化为 `struct usb_descriptor_header` 结构体指针和这里转化为 `struct usb_interface_descriptor` 结构体指针, 它就不再仅仅是一个地址, 而是代表了不同的含义。

326 行, `bDescriptorType` 等于 `USB_DT_INTERFACE` 并不说明它就一定是接口描述符了, 它的 `bLength` 还必须要等于 `USB_DT_INTERFACE_SIZE`。 `bLength` 和 `bDescriptorType` 一起才能决定一个描述符。

341 行到 352 行, 这几行首先要明白 `n`、`inu ms` 和 `nalts` 表示什么, `n` 记录的是接口的数目, 数组 `inu ms` 里的每一项都表示一个接口号, 数组 `nalts` 里的每一项记录的是每个接口拥有的设置数目, `inu ms` 和 `nalts` 两个数组里的元素是一一对应的, `inu ms[0]` 就对应 `nalts[0]`, `inu ms[1]` 就对应 `nalts[1]`。其次还要记住, 发送 `GET_DESCRIPTOR` 请求时, 设备并不一定会按照接口 1, 接口 2 这样的顺序循规蹈矩地返回数据。

361 行, `buffer` 的最后面可能会有一些垃圾数据, 为了去除这些垃圾数据, 这里需要将 `size` 和配置描述符里的 `wTotalLength` 修正一下。

364 行, 经过上面的循环之后, 如果统计得到的接口数目和配置描述符里的 `bNumInterfaces` 不符, 或者干脆就没有发现配置里有什么接口, 就警告一下。

373 行, 一个 `for` 循环, 目的是看一看是不是遗漏了哪个接口号, 比如说配置 6 个接口, 每个接口号都应该对应数组 `inu ms` 里的一项, 如果在 `inu ms` 里面没有发现这个接口号, 比如 2 吧, 那 2 这个接口号就神秘失踪了, 你找不到接口 2。这个当然也属于不规范的, 需要警告一下。

384 行, 又是一个 `for` 循环, `USB_MAXALTSETTING` 的定义在 `config.c` 里。

```
11 #define USB_MAXALTSETTING          128      /* Hard limit */
```

一个接口最多可以有 128 个设置。394 行根据每个接口拥有的设置数目为对应的 `intf_cache` 数组项申请内存。

403 行, 配置描述符后面紧跟的不一定就是接口描述符, 还可能是 `class`-和 `vendor-specific` 描述符。不管有没有, 先把 `buffer` 的地址赋给 `extra`, 如果没有扩展的描述符, 则 404 行返回的 `i` 就等于 0, `extralen` 也就为 0。

404 行, 调用 `find_next_descriptor()` 在 `buffer` 里寻找配置描述符后面跟着的第一个接口描述符。它也在 `config.c` 中定义, 进去看一看。

```
22 static int find_next_descriptor(unsigned char *buffer, int size,
23     int dt1, int dt2, int *num_skipped)
24 {
25     struct usb_descriptor_header *h;
26     int n = 0;
27     unsigned char *buffer0 = buffer;
28
29     /* Find the next descriptor of type dt1 or dt2 */
30     while (size > 0) {
31         h = (struct usb_descriptor_header *) buffer;
32         if (h->bDescriptorType == dt1 || h->bDescriptorType == dt2)
```

```

33         break;
34         buffer += h->bLength;
35         size -= h->bLength;
36         ++n;
37     }
38
39     /* Store the number of descriptors skipped and return the
40      * number of Bytes skipped */
41     if (num_skipped)
42         *num_skipped = n;
43     return buffer - buffer0;
44 }

```

这个函数需要传递两个描述符类型的参数，32 行已经清清楚楚地表明它不是专一地去寻找一种描述符，而是去寻找两种描述符，比如你指定 `dt1` 为 `USB_DT_INTERFACE`，`dt2` 为 `USB_DT_ENDPOINT` 时，只要能够找到接口描述符或端点描述符中的一个，这个函数就返回。`usb_parse_configuration` 函数在 404 行只需要寻找下一个接口描述符，所以 `dt1` 和 `dt2` 都设置为 `USB_DT_INTERFACE`。

这个函数结束后，`num_skipped` 里记录的是搜索过程中忽略的 `dt1` 和 `dt2` 之外其他描述符的数目，返回值表示搜索结束时，`buffer` 的位置比搜索开始时前进的字节数。其他没什么好讲的，还是回到 `usb_parse_configuration` 函数。

410 行，根据 `find_next_descriptor` 的结果修正 `buffer` 和 `size`。你可能对 C 语言里的按引用传递和按值传递已经烂熟于心，看到 `find_next_descriptor()` 那里传递的是 `buffer`，一个指针，条件反射地觉得它里面面对 `buffer` 的修改必定影响了外面的 `buffer`，所以认为 `buffer` 已经指向了寻找到的接口描述符。但是 `find_next_descriptor` 里修改的只是参数中 `buffer` 的值，并没有修改它指向的内容，对于地址本身来说仍然只能算是按值传递，怎么修改都影响不到函数外边，所以这里的 410 行仍然要对 `buffer` 的位置进行修正。

414 行，事不过三，三个 `for` 循环之后轮到了一个 `while` 循环。如果 `size` 大于 0，就说明配置描述符后面找到了一个接口描述符，根据这个接口描述符的长度，已经可以解析出一个完整的接口描述符了，但是仍然没到乐观时，这个接口描述符后面还会跟着一群端点描述符，再然后还会有其他接口描述符。

所以我们又迎来了另一个函数——`usb_parse_interface`，先不管它长什么样子，毕竟 `usb_parse_configuration()` 就快到头了，暂时只需要知道它返回时，`buffer` 的位置已经在下一个接口描述符那里了，同理，对 `buffer` 地址本身来说是按值传递的，所以 420 行要对这个位置和长度进行一下调整以适应新形势。那么这个 `while` 循环的意思就很明显了，对 `buffer` 一段一段地解析，直到再也找不到接口描述符了。

425 行，最后这个 `for` 循环没什么实质性的内容，就是找一下每个接口是不是有哪个设置编号给漏过去了，只要有耐心，你就能看得懂。接下来还是看 `config.c` 里的 `usb_parse_interface()` 函数。

```

158 static int usb_parse_interface(struct device *ddev, int cfgno,
159     struct usb_host_config *config, unsigned char *buffer, int size,
160     u8 inu ms[], u8 nalts[])
161 {
162     unsigned char *buffer0 = buffer;
163     struct usb_interface_descriptor *d;
164     int inum, asnum;
165     struct usb_interface_cache *intfc;
166     struct usb_host_interface *alt;
167     int i, n;
168     int len, retval;
169     int num_ep, num_ep_orig;
170
171     d = (struct usb_interface_descriptor *) buffer;
172     buffer += d->bLength;
173     size -= d->bLength;
174
175     if (d->bLength < USB_DT_INTERFACE_SIZE)
176         goto skip_to_next_interface_descriptor;
177
178     /* Which interface entry is this? */
179     intfc = NULL;
180     inum = d->bInterfaceNumber;
181     for (i = 0; i < config->desc.bNumInterfaces; ++i) {
182         if (inu ms[i] == inum) {
183             intfc = config->intf_cache[i];
184             break;
185         }
186     }
187     if (!intfc || intfc->num_altsetting >= nalts[i])
188         goto skip_to_next_interface_descriptor;
189
190     /* Check for duplicate altsetting entries */
191     asnum = d->bAlternateSetting;
192     for ((i = 0, alt = &intfc->altsetting[0]);
193         i < intfc->num_altsetting;
194         (++i, ++alt)) {
195         if (alt->desc.bAlternateSetting == asnum) {
196             dev_warn(ddev, "Duplicate descriptor for config %d "
197                 "interface %d altsetting %d, skipping\n",
198                 cfgno, inum, asnum);
199             goto skip_to_next_interface_descriptor;
200         }
201     }
202
203     ++intfc->num_altsetting;
204     memcpy(&alt->desc, d, USB_DT_INTERFACE_SIZE);
205
206     /* Skip over any Class Specific or Vendor Specific descriptors;
207      * find the first endpoint or interface descriptor */
208     alt->extra = buffer;
209     i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
210         USB_DT_INTERFACE, &n);
211     alt->extralen = i;
212     if (n > 0)
213         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
214             n, plural(n), "interface");
215     buffer += i;
216     size -= i;

```

```

217
218     /* Allocate space for the right(?) number of endpoints */
219     num_ep = num_ep_orig = alt->desc.bNumEndpoints;
220     alt->desc.bNumEndpoints = 0;           // Use as a counter
221     if (num_ep > USB_MAXENDPOINTS) {
222         dev_warn(ddev, "too many endpoints for config %d interface %d "
223             "altsetting %d: %d, using maximum allowed: %d\n",
224             cfgno, inum, asnum, num_ep, USB_MAXENDPOINTS);
225         num_ep = USB_MAXENDPOINTS;
226     }
227
228     if (num_ep > 0) {           /* Can't allocate 0 Bytes */
229         len = sizeof(struct usb_host_endpoint) * num_ep;
230         alt->endpoint = kzalloc(len, GFP_KERNEL);
231         if (!alt->endpoint)
232             return -ENOMEM;
233     }
234
235     /* Parse all the endpoint descriptors */
236     n = 0;
237     while (size > 0) {
238         if (((struct usb_descriptor_header *) buffer)->bDescriptorType
239             == USB_DT_INTERFACE)
240             break;
241         retval = usb_parse_endpoint(ddev, cfgno, inum, asnum, alt,
242             num_ep, buffer, size);
243         if (retval < 0)
244             return retval;
245         ++n;
246
247         buffer += retval;
248         size -= retval;
249     }
250
251     if (n != num_ep_orig)
252         dev_warn(ddev, "config %d interface %d altsetting %d has %d "
253             "endpoint descriptor%s, different from the interface "
254             "descriptor's value: %d\n",
255             cfgno, inum, asnum, n, plural(n), num_ep_orig);
256     return buffer - buffer0;
257
258 skip_to_next_interface_descriptor:
259     i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
260         USB_DT_INTERFACE, NULL);
261     return buffer - buffer0 + i;
262 }

```

171 行，传递过来的 `buffer` 里开头儿那部分只能是一个接口描述符，所以这里将地址转化为 `struct usb_interface_descriptor` 结构体指针，然后调整 `buffer` 的位置和 `size`。

175 行，“只能是”并不说明“它就是”，只有 `bLength` 等于 `USB_DT_INTERFACE_SIZE` 才说明 `USB_DT_INTERFACE_SIZE` 字节确实是一个接口描述符。否则就没必要再对这些数据进行什么处理了，直接跳到最后吧。先看一看这个函数最后都发生了什么，从新的位置开始再次调用 `find_next_descriptor()` 在 `buffer` 里寻找下一个接口描述符。



179 行，因为数组 `inu ms` 并不一定是按照接口的顺序来保存接口号的，`inu ms[1]` 对应的可能是接口 1 也可能是接口 0。所以这里要用 `for` 循环来寻找这个接口对应着 `inu ms` 里的哪一项，从而根据在数组里的位置获得接口对应的 `struct usb_interface_cache` 结构体。`usb_parse_configuration()` 已经告诉了我们，同一个接口在 `inu ms` 和 `intf_cache` 这两个数组里的位置是一样的。

191 行，获得这个接口描述符对应的设置编号，然后根据这个编号从接口的 `cache` 里搜索看这个设置是不是已经遇到过了。如果设置已经遇到过，就没必要再对这个接口描述符进行处理，直接跳到最后。否则就意味着发现了一个新的设置，要将它添加到 `cache` 里，并将 `cache` 里的设置数目 `num_altsetting` 加 1。要记住，设置是用 `struct usb_host_interface` 结构来表示的，一个接口描述符就对应一个设置。

208 行，这段代码很熟悉。现在 `buffer` 开头儿的接口描述符已经理清了，现在要解析它后面的那些数据了。先把位置赋给这个刚解析出来的接口描述符的 `extra`，然后再从这个位置开始去寻找下一个距离最近的一个接口描述符或端点描述符。如果这个接口描述符后面还跟有 `class-` 或 `vendor-specific` 描述符，则 `find_next_descriptor` 的返回值会大于 0，`buffer` 的位置和 `size` 也要进行相应调整，来指向新找到的接口描述符或端点描述符。

这里 `find_next_descriptor` 的 `dt1` 参数和 `dt2` 参数就不再是一样的了，因为如果一个接口只用到端点 0，它的接口描述符后面是不会跟有端点描述符的。

219 行，获得这个设置使用的端点数目，然后将相应接口描述符里的 `bNumEndpoints` 置 0，为什么？往下看，`USB_MAXENDPOINTS` 在 `config.c` 中定义的。

```
12 #define USB_MAXENDPOINTS          30          /* Hard limit */
```

为什么这个最大上限为 30？前面提到过，请参考前面章节的讲解。然后根据端点数为接口描述符里的 `endpoint` 数组申请内存。

237 行，走到这里，`buffer` 开头儿的那个接口描述符已经理清了，而且也找到了下一个接口描述符或端点描述符的位置，该从这个新的位置开始解析了，于是又遇到了一个似曾相识的 `while` 循环。

238 行，先判断一下前面找到的是接口描述符还是端点描述符，如果是接口描述符就中断这个 `while` 循环，返回与下一个接口描述符的距离。否则说明在 `buffer` 当前的位置上是一个端点描述符，因此就要迎来另一个函数 `usb_parse_endpoint` 对紧接着的数据进行解析。`usb_parse_endpoint()` 返回时，`buffer` 的位置已经在下一个端点描述符里了。247 行，调整 `buffer` 的位置长度，这个 `while` 循环也很明显了，对 `buffer` 一段一段地解析，直到遇到下一个接口描述符或者已经走到 `buffer` 结尾。现在看一看 `config.c` 中定义的 `usb_parse_endpoint` 函数。

```
46 static int usb_parse_endpoint(struct device *ddev, int cfgno, int inum,
47     int asnum, struct usb_host_interface *ifp, int num_ep,
```

```

48 unsigned char *buffer, int size)
49 {
50     unsigned char *buffer0 = buffer;
51     struct usb_endpoint_descriptor *d;
52     struct usb_host_endpoint *endpoint;
53     int n, i, j;
54
55     d = (struct usb_endpoint_descriptor *) buffer;
56     buffer += d->bLength;
57     size -= d->bLength;
58
59     if (d->bLength >= USB_DT_ENDPOINT_AUDIO_SIZE)
60         n = USB_DT_ENDPOINT_AUDIO_SIZE;
61     else if (d->bLength >= USB_DT_ENDPOINT_SIZE)
62         n = USB_DT_ENDPOINT_SIZE;
63     else {
64         dev_warn(ddev, "config %d interface %d altsetting %d has an "
65                 "invalid endpoint descriptor of length %d, skipping\n",
66                 cfgno, inum, asnum, d->bLength);
67         goto skip_to_next_endpoint_or_interface_descriptor;
68     }
69
70     i = d->bEndpointAddress & ~USB_ENDPOINT_DIR_MASK;
71     if (i >= 16 || i == 0) {
72         dev_warn(ddev, "config %d interface %d altsetting %d has an "
73                 "invalid endpoint with address 0x%X, skipping\n",
74                 cfgno, inum, asnum, d->bEndpointAddress);
75         goto skip_to_next_endpoint_or_interface_descriptor;
76     }
77
78     /* Only store as many endpoints as we have room for */
79     if (ifp->desc.bNumEndpoints >= num_ep)
80         goto skip_to_next_endpoint_or_interface_descriptor;
81
82     endpoint = &ifp->endpoint[ifp->desc.bNumEndpoints];
83     ++ifp->desc.bNumEndpoints;
84
85     memcpy(&endpoint->desc, d, n);
86     INIT_LIST_HEAD(&endpoint->urb_list);
87
88     /* If the bInterval value is outside the legal range,
89      * set it to a default value: 32 ms */
90     i = 0; /* i = min, j = max, n = default */
91     j = 255;
92     if (usb_endpoint_xfer_int(d)) {
93         i = 1;
94         switch (to_usb_device(ddev)->speed) {
95             case USB_SPEED_HIGH:
96                 n = 9; /* 32 ms = 2^(9-1) uframes */
97                 j = 16;
98                 break;
99             default: /* USB_SPEED_FULL or _LOW */
100                 /* For low-speed, 10 ms is the official minimum.
101                  * But some "overclocked" devices might want faster
102                  * polling so we'll allow it. */
103                 n = 32;
104                 break;
105         }
106     } else if (usb_endpoint_xfer_isoc(d)) {

```

```

107         i = 1;
108         j = 16;
109         switch (to_usb_device(ddev)->speed) {
110             case USB_SPEED_HIGH:
111                 n = 9;          /* 32 ms = 2^(9-1) uframes */
112                 break;
113             default:
114                 n = 6;          /* 32 ms = 2^(6-1) frames */
115                 break;
116         }
117     }
118     if (d->bInterval < i || d->bInterval > j) {
119         dev_warn(ddev, "config %d interface %d altsetting %d "
120             "endpoint 0x%X has an invalid bInterval %d, "
121             "changing to %d\n",
122             cfgno, inum, asnum,
123             d->bEndpointAddress, d->bInterval, n);
124         endpoint->desc.bInterval = n;
125     }
126
127     /* Skip over any Class Specific or Vendor Specific descriptors;
128      * find the next endpoint or interface descriptor */
129     endpoint->extra = buffer;
130     i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
131         USB_DT_INTERFACE, &n);
132     endpoint->extralen = i;
133     if (n > 0)
134         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
135             n, plural(n), "endpoint");
136     return buffer - buffer0 + i;
137
138 skip_to_next_endpoint_or_interface_descriptor:
139     i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
140         USB_DT_INTERFACE, NULL);
141     return buffer - buffer0 + i;
142 }

```

55 行，`buffer` 开头儿只能是一个端点描述符，所以这里将地址转化为 `struct usb_endpoint_descriptor` 结构体指针，然后调整 `buffer` 的位置和 `size`。

59 行，这里要明白的是端点描述符与配置描述符、接口描述符不一样，它是可能有两种大小的。

70 行，得到端点号。这里的端点号不能为 0，因为端点 0 是没有描述符的，也不能大于 16。

79 行，这个 `bNumEndpoints` 在 `usb_parse_interface()` 的 220 行是被赋为 0 值了的。

82 行，这个 `endpoint` 数组在 `usb_parse_interface()` 的 230 行也是已经申请好内存了的。从这里你应该明白 `bNumEndpoints` 是被当成了一个计数器，发现一个端点描述符，它就加 1，并把找到的端点描述符复制到设置的 `endpoint` 数组里。

86 行，初始化端点的 `urb` 队列 `urb_list`。

88 行到 125 行，这堆代码的目的是处理端点的 `bInterval`，先得明白几个问题。第一个问题就是 *i*, *j*, *n* 分别表示什么。90 行到 117 行这么多行的代码就为了给它们选择一个合适的值，*i* 和 *j* 限定了 `bInterval` 的一个范围，`bInterval` 如果在这里面，像 124 行做的那样将 *n* 赋给它，那么 *n* 表示的就是 `bInterval` 的一个默认值。*i* 和 *j* 的默认值分别为 0 和 255，也就是说合法的范围默认是 0~255。对于批量端点和控制端点，`bInterval` 对我们来说并没有太大的用处，不过协议中还是规定了，这个范围只能为 0~255。对于中断端点和等时端点，`bInterval` 表演的舞台就很大了，对这个范围也要做一些调整。

第二个问题就是如何判断端点是中断的还是等时的。这涉及两个函数 `usb_endpoint_xfer_int` 和 `usb_endpoint_xfer_isoc`，它们都在 `include/linux/usb.h` 中定义。

```

596     static inline int usb_endpoint_xfer_int(const struct
usb_endpoint_descriptor *epd)
597 {
598     return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
599             USB_ENDPOINT_XFER_INT);
600 }
601
609     static inline int usb_endpoint_xfer_isoc(const struct
usb_endpoint_descriptor *epd)
610 {
611     return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
612             USB_ENDPOINT_XFER_ISOC);
613 }

```

这两个函数的意思简单明了，另外两个函数就是 `usb_endpoint_xfer_bulk` 和 `usb_endpoint_xfer_control`，用来判断批量端点和控制端点的。

第三个问题是 `to_usb_device`。`usb_parse_endpoint()` 的参数是 `struct device` 结构体，要获得设备的速度就需要使用 `to_usb_device` 将它转化为 `struct usb_device` 结构体，这是一个 `include/linux/usb.h` 中定义的宏：

```

410 #define to_usb_device(d) container_of(d, struct usb_device, dev)

```

接着看 `usb_parse_endpoint` 的 129 行，现在你对这几行的意思明白了。这里接着在 `buffer` 里寻找下一个端点描述符或者接口描述符。

经过 `usb_parse_configuration`、`usb_parse_interface` 和 `usb_parse_endpoint` 这三个函数的层层推进，通过 `GET_DESCRIPTOR` 请求所获得那堆数据现在已经解析地清清楚楚。现在，设备的各个配置信息已经了然于胸，那接下来设备的那条生命线该怎么去走？它已经可以进入 `Configured` 状态了吗？事情没这么简单，光是获得设备各个配置信息没用，要进入 `Configured` 状态，你还得有选择、有目的、有步骤、有计划地去配置设备，那怎么才做到？这好像就不是 `Core` 能够答复的问题了，毕竟它并不知道你希望设备采用哪种配置，只有设备的驱动才知道，所以接下来设备要做的是去在设备模型中寻找属于自己的驱动。

不要忘记设备的 `struct usb_device` 结构体在出生时就带有 `usb_bus_type` 和 `usb_device_type` 这样的“胎记”，Linux 设备模型根据总线类型 `usb_bus_type` 将设备添加到 USB 总线的那条有名的设备链表里，然后去轮询 USB 总线的另外一条有名的驱动链表，针对每个找到的驱动去调用 USB 总线的 `match` 函数，也就是 `usb_device_match()` 函数，去为设备寻找另一个匹配的驱动。`match` 函数会根据设备的自身条件和类型 `usb_device_type` 安排设备走设备那条路，从而匹配到那个对所有 `usb_device_type` 类型的设备都来者不拒的“花心大萝卜”，USB 世界里唯一的一个 USB 设备驱动（不是 USB 接口驱动）`struct device_driver` 结构体对象 `usb_generic_driver`。

---

## 31. 驱动的生命线（一）

`usb_generic_driver` 是一个 USB 设备都会争先恐后地找它配对儿。

现在开始就沿着 `usb_generic_driver` 的成名之路走一走，设备的生命线你可以想当然地认为是从你的 USB 设备连接到 Hub 的某个端口时开始，驱动的生命线就必须得回溯到 USB 子系统的初始化函数 `usb_register_device_driver`。

```
895  retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
896  if (!retval)
897      goto out;
```

在 USB 子系统初始化时，就调用了 `driver.c` 里面的 `usb_register_device_driver` 函数将 `usb_generic_driver` 注册给系统了，下面来看一看。

```
671 int usb_register_device_driver(struct usb_device_driver *new_udriver,
672                                struct module *owner)
673 {
674     int retval = 0;
675
676     if (usb_disabled())
677         return -ENODEV;
678
679     new_udriver->drvwrap.for_devices = 1;
680     new_udriver->drvwrap.driver.name = (char *) new_udriver->name;
681     new_udriver->drvwrap.driver.bus = &usb_bus_type;
682     new_udriver->drvwrap.driver.probe = usb_probe_device;
683     new_udriver->drvwrap.driver.remove = usb_unbind_device;
684     new_udriver->drvwrap.driver.owner = owner;
685
686     retval = driver_register(&new_udriver->drvwrap.driver);
687
688     if (!retval) {
689         pr_info("%s: registered new device driver %s\n",
690               usbcore_name, new_udriver->name);
691         usbfs_update_special();
692     } else {
693         printk(KERN_ERR "%s: error %d registering device "
```

```

694         "        driver %s\n",
695         usbcore_name, retval, new_udriver->name);
696     }
697
698     return retval;
699 }

```

676 行，判断一下 USB 子系统是不是在你启动内核时就被禁止了，如果是的话，它的生命也就太短暂了。

679 行，`for_devices` 就是在这儿被初始化为 1 的，有了它，`match` 里的 `is_usb_device_driver` 才有章可循，有凭可依。

下面就是充实了一下 `usb_generic_driver` 里嵌入的 `struct device_driver` 结构体，`usb_generic_driver` 就是通过它和设备模型连上关系的。`name` 就是 `usb_generic_driver` 的名字，即 USB，所属的总线类型同样被设置为 `usb_bus_type`，然后是指定 `probe` 函数和 `remove` 函数。

686 行，调用设备模型的函数 `driver_register` 将 `usb_generic_driver` 添加到 USB 总线的那条驱动链表里。

`usb_generic_driver` 和 USB 设备匹配成功后，就会调用 682 行指定的 `probe` 函数 `usb_probe_device()`，现在看一看 `driver.c` 中定义的这个函数。

```

152 static int usb_probe_device(struct device *dev)
153 {
154     struct usb_device_driver *udriver =
155         to_usb_device_driver(dev->driver);
156     struct usb_device *udev;
157     int error = -ENODEV;
158
159     dev_dbg(dev, "%s\n", __FUNCTION__);
160     if (!is_usb_device(dev)) /* Sanity check */
161         return error;
162
163     udev = to_usb_device(dev);
164
165     /* TODO: Add real matching code */
166
167     /* The device should always appear to be in use
168      * unless the driver supports autosuspend.
169      */
170     udev->pm_usage_cnt = !(udriver->supports_autosuspend);
171
172     error = udriver->probe(udev);
173     return error;
174 }

```

154 行，`to_usb_device_driver` 是 `include/linux/usb.h` 中定义的一个宏，和前面遇到的那个 `to_usb_device` 有异曲同工之妙。

```

889 #define to_usb_device_driver(d) container_of(d, struct usb_device_driver,
\
890          drvwrap.driver)

```

160 行, `match` 函数 543 行的函数又调到这儿来“把门儿”了, 如果你这个设备的类型不是 `usb_device_type`, 那怎么从前面走到这儿的它管不着, 但是到它这里就不可能再蒙混过关了。你的设备虽然和 `usb_generic_driver` 是配对成功了, 但是还要经过判断。

163 行, 前面刚提到 `to_usb_device` 就到这里来了。

170 行, `pm_usage_cnt` 和 `supports_autosuspend` 这两个前面都提到过一下。每个 `struct usb_interface` 或 `struct usb_device` 里都有一个 `pm_usage_cnt`, 每个 `struct usb_driver` 或 `struct usb_device_driver` 里都有一个 `supports_autosuspend`。只有 `pm_usage_cnt` 为 0 时才会允许接口 `autosuspend`, 如果 `supports_autosuspend` 为 0 就不再允许绑定到这个驱动的设备 `autosuspend`。

接口? 设备? 需要时, 接口是接口设备是设备; 不需要时, 接口设备一个样。这里就是不需要时, 所以将上面的接口换成设备套一下就是: `pm_usage_cnt` 为 0 时才会允许设备 `autosuspend`, `supports_autosuspend` 为 0, 就不再允许绑定到这个驱动的设备 `autosuspend`。

所有的 USB 设备都是绑定到 `usb_generic_driver` 上面的, `usb_generic_driver` 的 `supports_autosuspend` 字段又是为 1 的, 所以这行就是将设备 `struct usb_device` 结构体的 `pm_usage_cnt` 置为了 0, 也就是说允许设备 `autosuspend`。但是不是说将 `pm_usage_cnt` 轻轻松松置为 0, 设备就能够 `autosuspend` 了, 驱动必须得实现一对 `suspend/resume` 函数供 PM 子系统驱使, `usb_generic_driver` 里的这对函数就是 `generic_suspend/generic_resume`。

172 行, 这里要调用 `usb_generic_driver` 自己私有的 `probe` 函数 `generic_probe()` 对你的设备进行进一步的审查, 它在 `generic.c` 中定义。

```

153 static int generic_probe(struct usb_device *udev)
154 {
155     int err, c;
156
157     /* put device-specific files into sysfs */
158     usb_create_sysfs_dev_files(udev);
159
160     /* Choose and set the configuration. This registers the interfaces
161      * with the driver core and lets interface drivers bind to them.
162      */
163     c = choose_configuration(udev);
164     if (c >= 0) {
165         err = usb_set_configuration(udev, c);
166         if (err) {
167             dev_err(&udev->dev, "can't set config %d, error %d\n",
168                     c, err);
169             /* This need not be fatal. The user can try to
170              * set other configurations. */
171         }
172     }
173 }

```

```

174     /* USB device state == configured ... usable */
175     usb_notify_add_device(udev);
176
177     return 0;
178 }

```

这函数说简单也简单，说复杂也复杂，简单的是外表，复杂的是内心。用一句话去概括，就是从设备众多配置中选择一个合适的，然后去配置设备，从而让设备进入期待已久的 Configured 状态。先看一看是怎么选择一个配置的，调用的是 generic.c 里的 choose\_configuration 函数。

```

42 static int choose_configuration(struct usb_device *udev)
43 {
44     int i;
45     int num_configs;
46     int insufficient_power = 0;
47     struct usb_host_config *c, *best;
48
49     best = NULL;
50     c = udev->config;
51     num_configs = udev->descriptor.bNumConfigurations;
52     for (i = 0; i < num_configs; (i++, c++)) {
53         struct usb_interface_descriptor *desc = NULL;
54
55         /* It's possible that a config has no interfaces! */
56         if (c->desc.bNumInterfaces > 0)
57             desc = &c->intf_cache[0]->altsetting->desc;
58
59         /*
60          * HP's USB bus-powered keyboard has only one configuration
61          * and it claims to be self-powered; other devices may have
62          * similar errors in their descriptors. If the next test
63          * were allowed to execute, such configurations would always
64          * be rejected and the devices would not work as expected.
65          * In the meantime, we run the risk of selecting a config
66          * that requires external power at a time when that power
67          * isn't available. It seems to be the lesser of two evils.
68          *
69          * Bugzilla #6448 reports a device that appears to crash
70          * when it receives a GET_DEVICE_STATUS request! We don't
71          * have any other way to tell whether a device is self-powered,
72          * but since we don't use that information anywhere but here,
73          * the call has been removed.
74          *
75          * Maybe the GET_DEVICE_STATUS call and the test below can
76          * be reinstated when device firmwares become more reliable.
77          * Don't hold your breath.
78          */
79 #if 0
80         /* Rule out self-powered configs for a bus-powered device */
81         if (bus_powered && (c->desc.bmAttributes &
82                             USB_CONFIG_ATT_SELFPOWER))
83             continue;
84 #endif
85
86         /*
87          * The next test may not be as effective as it should be.

```



```

88      * Some hubs have errors in their descriptor, claiming
89      * to be self-powered when they are really bus-powered.
90      * We will overestimate the amount of current such hubs
91      * make available for each port.
92      *
93      * This is a fairly benign sort of failure. It won't
94      * cause us to reject configurations that we should have
95      * accepted.
96      */
97
98      /* Rule out configs that draw too much bus current */
99      if (c->desc.bMaxPower * 2 > udev->bus_mA) {
100          insufficient_power++;
101          continue;
102      }
103
104      /* When the first config's first interface is one of Microsoft's
105       * pet nonstandard Ethernet-over-USB protocols, ignore it unless
106       * this kernel has enabled the necessary host side driver.
107       */
108      if (i == 0 && desc && (is_rndis(desc) || is_activesync(desc))) {
109          #if !defined(CONFIG_USB_NET_RNDIS_HOST)
110              continue;
111          #else
112              best = c;
113          #endif
114      }
115
116      /* From the remaining configs, choose the first one whose
117       * first interface is for a non-vendor-specific class.
118       * Reason: Linux is more likely to have a class driver
119       * than a vendor-specific driver. */
120      else if (udev->descriptor.bDeviceClass !=
121               USB_CLASS_VENDOR_SPEC &&
122               (!desc || desc->bInterfaceClass !=
123                USB_CLASS_VENDOR_SPEC)) {
124          best = c;
125          break;
126      }
127
128      /* If all the remaining configs are vendor-specific,
129       * choose the first one. */
130      else if (!best)
131          best = c;
132  }
133
134  if (insufficient_power > 0)
135      dev_info(&udev->dev, "rejected %d configuration%s "
136              "due to insufficient available bus power\n",
137              insufficient_power, plural(insufficient_power));
138
139  if (best) {
140      i = best->desc.bConfigurationValue;
141      dev_info(&udev->dev,
142              "configuration #%d chosen from %d choice%s\n",
143              i, num_configs, plural(num_configs));
144  } else {
145      i = -1;

```

```

146         dev_warn(&udev->dev,
147                 "no configuration chosen from %d choice%s\n",
148                 num_configs, plural(num_configs));
149     }
150     return i;
151 }

```

设备各个配置的详细信息在设备自身的漫漫人生旅途中就已经获取并存放在相关的几个成员里了，怎么从中挑选一个让人满意的配置？显然谁都会说去一个一个地浏览每个配置，查看有没有称心如意的，于是就有了 52 行的 for 循环。

刚看到这个 for 循环就有点儿傻眼了，居然注释要远远多于代码，我们把这个 for 循环分成三大段，59 行到 84 行这一段，你什么都可以不看，就是不能不看那个 `#if 0`，一见到它，就意味着你可以略过这么一大段代码了。

第二段是 98 行到 102 行，这一段牵扯到最让人无可奈何的一对矛盾，索取与给予。一个配置索取的电流比 Hub 所能给予的还要大，显然它不会是一个让人满意的配置。

第三段是 108 行到 131 行，关于这段只说一个原则，Linux 更喜欢那些标准的东西，比如 `USB_CLASS_VIDEO`、`USB_CLASS_AUDIO` 等这样子的设备和接口就更讨人喜欢一些，所以就会优先选择非 `USB_CLASS_VENDOR_SPEC` 的接口。

for 循环之后，剩下的那些部分都是调试用的，输出一些调试信息，不需要去关心，不过里面出现了一个有趣的函数 `plural`，它是一个在 `generic.c` 开头儿定义的内联函数。

```

23 static inline const char *plural(int n)
24 {
25     return (n == 1 ? "" : "s");
26 }

```

参数 `n` 为 1 返回一个空字符串，否则返回一个“s”，看一下使用了这个函数的打印语句，就明白它是用来打印一个英语名词的单复数的，复数的话就加上一个“s”。

不管你疑惑也好，满意也好，`choose_configuration` 就是这样按照自己的标准挑选了一个比较合自己心意的配置，接下来当然就是要用这个配置去配置设备以便让它迈进 `Configured` 状态了。

## 32. 驱动的生命线（二）

Core 配置设备使用的是 `message.c` 里的 `usb_set_configuration` 函数。

```

1430 int usb_set_configuration(struct usb_device *dev, int configuration)
1431 {

```

```

1432     int i, ret;
1433     struct usb_host_config *cp = NULL;
1434     struct usb_interface **new_interfaces = NULL;
1435     int n, nintf;
1436
1437     if (configuration == -1)
1438         configuration = 0;
1439     else {
1440         for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {
1441             if (dev->config[i].desc.bConfigurationValue ==
1442                 configuration) {
1443                 cp = &dev->config[i];
1444                 break;
1445             }
1446         }
1447     }
1448     if ((!cp && configuration != 0))
1449         return -EINVAL;
1450
1451     /* The USB spec says configuration 0 means unconfigured.
1452      * But if a device includes a configuration numbered 0,
1453      * we will accept it as a correctly configured state.
1454      * Use -1 if you really want to unconfigure the device.
1455      */
1456     if (cp && configuration == 0)
1457         dev_warn(&dev->dev, "config 0 descriptor??\n");
1458
1459     /* Allocate memory for new interfaces before doing anything else,
1460      * so that if we run out then nothing will have changed. */
1461     n = nintf = 0;
1462     if (cp) {
1463         nintf = cp->desc.bNumInterfaces;
1464         new_interfaces = kmalloc(nintf * sizeof(*new_interfaces),
1465                                 GFP_KERNEL);
1466         if (!new_interfaces) {
1467             dev_err(&dev->dev, "Out of memory");
1468             return -ENOMEM;
1469         }
1470
1471         for (; n < nintf; ++n) {
1472             new_interfaces[n] = kzalloc(
1473                 sizeof(struct usb_interface),
1474                 GFP_KERNEL);
1475             if (!new_interfaces[n]) {
1476                 dev_err(&dev->dev, "Out of memory");
1477                 ret = -ENOMEM;
1478                 free_interfaces:
1479                 while (--n >= 0)
1480                     kfree(new_interfaces[n]);
1481                 kfree(new_interfaces);
1482                 return ret;
1483             }
1484         }
1485
1486         i = dev->bus_mA - cp->desc.bMaxPower * 2;
1487         if (i < 0)
1488             dev_warn(&dev->dev, "new config #%d exceeds power "
1489                     "limit by %dmA\n",
1490                     configuration, -i);

```

```

1491     }
1492
1493     /* Wake up the device so we can send it the Set-Config request */
1494     ret = usb_autoresume_device(dev);
1495     if (ret)
1496         goto free_interfaces;
1497
1498     /* if it's already configured, clear out old state first.
1499      * getting rid of old interfaces means unbinding their drivers.
1500      */
1501     if (dev->state != USB_STATE_ADDRESS)
1502         usb_disable_device (dev, 1);    // Skip ep0
1503
1504     if ((ret = usb_control_ msg(dev, usb_sndctrlpipe(dev, 0),
1505                                USB_REQ_SET_CONFIGURATION, 0, configuration, 0,
1506                                NULL, 0, USB_CTRL_SET_TIMEOUT)) < 0) {
1507
1508         /* All the old state is gone, so what else can we do?
1509          * The device is probably useless now anyway.
1510          */
1511         cp = NULL;
1512     }
1513
1514     dev->actconfig = cp;
1515     if (!cp) {
1516         usb_set_device_state(dev, USB_STATE_ADDRESS);
1517         usb_autosuspend_device(dev);
1518         goto free_interfaces;
1519     }
1520     usb_set_device_state(dev, USB_STATE_CONFIGURED);
1521
1522     /* Initialize the new interface structures and the
1523      * hc/hcd/usbcore interface/endpoint state.
1524      */
1525     for (i = 0; i < nintf; ++i) {
1526         struct usb_interface_cache *intfc;
1527         struct usb_interface *intf;
1528         struct usb_host_interface *alt;
1529
1530         cp->interface[i] = intf = new_interfaces[i];
1531         intfc = cp->intf_cache[i];
1532         intf->altsetting = intfc->altsetting;
1533         intf->num_altsetting = intfc->num_altsetting;
1534         kref_get(&intfc->ref);
1535
1536         alt = usb_altnum_to_altsetting(intf, 0);
1537
1538         /* No altsetting 0? We'll assume the first altsetting.
1539          * We could use a GetInterface call, but if a device is
1540          * so non-compliant that it doesn't have altsetting 0
1541          * then I wouldn't trust its reply anyway.
1542          */
1543         if (!alt)
1544             alt = &intf->altsetting[0];
1545
1546         intf->cur_altsetting = alt;
1547         usb_enable_interface(dev, intf);
1548         intf->dev.parent = &dev->dev;
1549         intf->dev.driver = NULL;

```

```

1550     intf->dev.bus = &usb_bus_type;
1551     intf->dev.type = &usb_if_device_type;
1552     intf->dev.dma_mask = dev->dev.dma_mask;
1553     device_initialize (&intf->dev);
1554     mark_quiesced(intf);
1555     sprintf (&intf->dev.bus_id[0], "%d-%s:%d.%d",
1556             dev->bus->busnum, dev->devpath,
1557             configuration, alt->desc.bInterfaceNumber);
1558 }
1559 kfree(new_interfaces);
1560
1561 if (cp->string == NULL)
1562     cp->string = usb_cache_string(dev, cp->desc.iConfiguration);
1563
1564 /* Now that all the interfaces are set up, register them
1565  * to trigger binding of drivers to interfaces. probe()
1566  * routines may install different altsettings and may
1567  * claim() any interfaces not yet bound. Many class drivers
1568  * need that: CDC, audio, video, etc.
1569  */
1570 for (i = 0; i < nintf; ++i) {
1571     struct usb_interface *intf = cp->interface[i];
1572
1573     dev_dbg (&dev->dev,
1574             "adding %s (config #%d, interface %d)\n",
1575             intf->dev.bus_id, configuration,
1576             intf->cur_altsetting->desc.bInterfaceNumber);
1577     ret = device_add (&intf->dev);
1578     if (ret != 0) {
1579         dev_err(&dev->dev, "device_add(%s) --> %d\n",
1580                 intf->dev.bus_id, ret);
1581         continue;
1582     }
1583     usb_create_sysfs_intf_files (intf);
1584 }
1585
1586 usb_autosuspend_device(dev);
1587 return 0;
1588 }

```

这个函数可以分成三个部分，从 1432 行到 1491 行的这几十行是准备阶段，做常规检查，申请内存。1498 行到 1520 行这部分可是重头戏，就是在这里设备从 **Address** 发展到了 **Configured**。1522 行到 1584 行这个阶段也挺重要的，主要充实设备的每个接口并提交给设备模型，为它们寻找命中注定的接口驱动，过了这个阶段，**usb\_generic\_driver** 也就彻底从设备那儿得到满足了，**generic\_probe** 的历史使命也就完成了。

先看第一阶段，1437 行，**configuration** 是从前面的 **choose\_configuration()** 那里返回来的值，找到合适的配置的话，就返回那个配置的 **bConfigurationValue** 值，没有找到合适的配置的话，就返回 -1，所以这里的 **configuration** 值就可能有两种情况，或者为 -1，或者为配置的 **bConfigurationValue** 值。

当 **configuration** 为 -1 时，这里为什么又要把它改为 0 呢？要知道 **configuration** 这个值是要在后面的高潮阶段里发送 **SET\_CONFIGURATION** 请求时用的，关于 **SET\_CONFIGURATION**

请求，spec 里说，这个值必须为 0 或者与配置描述符的 `bConfigurationValue` 一致。如果为 0，则设备收到 `SET_CONFIGURATION` 请求后，仍然会待在 `Address` 状态。这里当 `configuration` 为 -1 也就是没有发现满意的配置时，设备不能进入 `Configured`，所以要把 `configuration` 的值改为 0，以便满足 `SET_CONFIGURATION` 请求的要求。

那接下来的问题就出来了，在没有找到合适配置时直接给 `configuration` 这个参数设置为 0，也就是让 `choose_configuration()` 返回 0 不就得了，干吗还这么麻烦先返回个 -1 再把它改成 0？有些设备就是有拿 0 当配置 `bConfigurationValue` 值的毛病，你又不让它用。想让设备回到 `Address` 状态时，`usb_set_configuration()` 就不传递 0 了，传递一个 -1，然后进行处理。如果 `configuration` 值为 0 或大于 0 的值，就从设备 `struct usb_device` 结构体的 `config` 数组里将相应配置的描述信息，也就是 `struct usb_host_config` 结构体给取出来。

1448 行，如果没有拿到配置的内容，`configuration` 值就必须为 0 了，让设备待在 `Address` 那里别动。这也很好理解，配置的内容都找不到了，还配置什么呢？当然，如果拿到了配置的内容，但同时 `configuration` 为 0，这就是对应了上面说的那种有毛病的设备的情况，就提出警告出现不正常现象了。

1461 行，过了这个 `if` 函数，第一阶段就告结束了。如果配置是实实在在存在的，就为它使用的接口都准备一个 `struct usb_interface` 结构体。`new_interfaces` 是开头儿就定义好的一个 `struct usb_interface` 结构体指针数组，数组的每一项都指向了一个 `struct usb_interface` 结构体，所以这里申请内存也要分两步走，先申请指针数组的，再申请每一项的。

## 33. 驱动的生命线（三）

现在查看第二阶段的重头戏，查看设备是怎么从 `Address` 进入 `Configured` 的。1501 行，如果已经处于 `Configured` 状态了，则退回到 `Address` 状态。仔细研究一下 `message.c` 里的 `usb_disable_device` 函数。

```

1034 void usb_disable_device(struct usb_device *dev, int skip_ep0)
1035 {
1036     int i;
1037
1038     dev_dbg(&dev->dev, "%s nuking %s URBs\n", __FUNCTION__,
1039           skip_ep0 ? "non-ep0" : "all");
1040     for (i = skip_ep0; i < 16; ++i) {
1041         usb_disable_endpoint(dev, i);
1042         usb_disable_endpoint(dev, i + USB_DIR_IN);
1043     }
1044     dev->toggle[0] = dev->toggle[1] = 0;
1045
1046     /* getting rid of interfaces will disconnect

```

```

1047     * any drivers bound to them (a key side effect)
1048     */
1049     if (dev->actconfig) {
1050         for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1051             struct usb_interface    *interface;
1052
1053             /* remove this interface if it has been registered */
1054             interface = dev->actconfig->interface[i];
1055             if (!device_is_registered(&interface->dev))
1056                 continue;
1057             dev_dbg (&dev->dev, "unregistering interface %s\n",
1058                     interface->dev.bus_id);
1059             usb_remove_sysfs_intf_files(interface);
1060             device_del (&interface->dev);
1061         }
1062
1063         /* Now that the interfaces are unbound, nobody should
1064          * try to access them.
1065          */
1066         for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1067             put_device (&dev->actconfig->interface[i]->dev);
1068             dev->actconfig->interface[i] = NULL;
1069         }
1070         dev->actconfig = NULL;
1071         if (dev->state == USB_STATE_CONFIGURED)
1072             usb_set_device_state(dev, USB_STATE_ADDRESS);
1073     }
1074 }

```

经过研究我们可以发现，`usb_disable_device` 函数的工作主要有两部分，一是将设备中所有端点给删除掉，另一个是将设备当前配置使用的每个接口都从系统里给 `unregister` 掉，也就是将接口和它对应的驱动给分开。

先说一下第二部分的工作，1049 行，`actconfig` 表示设备当前激活的配置，只有它不为空时才有接下来清理的必要。

1050 行到 1061 行这个 `for` 循环就是将这个配置的每个接口从设备模型的体系中删除掉，将它们和对应的接口驱动分开，没有驱动了，这些接口也就丧失了能力，当然也就什么作用都发挥不了了，这也是名字里那个 `disable` 的真正含义所在。

1066 行到 1070 行，将 `actconfig` 的 `interface` 数组置为空，然后再将 `actconfig` 置为空，这里你可能会有的一个疑问，为什么只是置为空，既然要清理 `actconfig`，为什么不直接将它占用的内存给释放掉？你应该注意到 `actconfig` 只是一个指针，一个地址，你应该首先弄清楚这个地址里保存的是什么东西再决定是不是将它给释放掉，那这个指针指向哪儿？它指向设备 `struct usb_device` 结构的 `config` 数组里的其中一项，当前被激活的是哪一个配置，它就指向 `config` 数组里的哪一项。你这里只是不想让设备当前激活任何一个配置而已，没必要将 `actconfig` 指向的那个配置给释放掉。

那另一个问题就出来了，既然 `actconfig` 指向了 `config` 里的一项，那为什么要把 `interface`

数组给置为空，这不是修改了配置的内容，从而也修改了 `config` 数组的内容吗？先别着急，在设备生命线那里取配置描述符，解析返回的那堆数据时，只是把每个配置里的 `cache` 数组，也就是 `intf_cache` 数组初始化了，并没有为 `interface` 数组充实任何的内容，这里做清理工作的目的就是要恢复原状，当然要将它置为空了。那么配置的 `interface` 数组又在哪里被充实了呢？在 `usb_set_configuration` 函数中第二个高潮阶段之后不是还有个第三个阶段呢，就在那里，激活了哪个配置，就为哪个配置的 `interface` 数组，填了点儿东西。

1071 行，如果这个设备此时确实是在 `CONFIGURED` 状态，就让它回到 `Address`。

现在回头来说说第一部分的清理工作。这个部分主要就是为每个端点调用了 `usb_disable_endpoint` 函数，将挂在它们上面的 `urb` 给取消掉。为什么要这么做？你想想，能调用到 `usb_disable_device` 这个函数，一般来说设备的状态要发生变化了，设备的状态都改变了，那设备的端点状态是不是也要改变？还有挂在它们上面的那些 `urb` 需不需要给取消掉？这些是很显然的事情，就拿现在让设备从 `Configured` 回到 `Address` 来说吧，在 `Address` 时，你只能通过默认管道也就是端点 0 对应的管道与设备进行通信的，但是在“`Configured`”时，设备的所有端点都是能够使用的，它们上面可能已经挂了一些 `urb` 正在处理或者将要处理，那么这时让设备要从 `CONFIGURED` 变到 `ADDRESS`，是不是应该先将这些 `urb` 给取消掉？

参数 `skip_ep0` 是什么意思？这里 `for` 循环的 `i` 从 `skip_ep0` 开始算起，也就是说 `skip_ep0` 为 1 的话，就不需要对端点 0 调用 `usb_disable_endpoint` 函数了。按常理来说，设备状态改变了，是需要把每个端点上面的 `urb` 取消掉的，这里面当然也要包括端点 0。但是写代码的人在这里创建出一个 `skip_ep0` 自然有他们的玄机，`usb_set_configuration()`调用这个函数时参数 `skip_ep0` 的值是什么？是 1，因为这时候是从 `Configured` 回到 `Address`，这个过程中，其他端点是从能够使用变成了不能使用，但端点 0 却是一直都很强势，虽说是设备发生了状态的变化，但在这两个状态里它都是要正常使用的，所以就没必要 `disable` 它了。

什么时候需要 `disable` 端点 0？在目前版本的内核中我只发现了两种情况，一种是设备要断开时，另一种是设备从 `Default` 进化到 `Address` 时。虽说不管是 `Default` 还是 `Address`，端点 0 都是需要能够正常使用的，但因为地址发生了变化了，毫无疑问，你需要将挂在它上面的 `urb` 清除掉。当时讲设备生命线时，在设置完设备地址，设备进入 `Address` 后，第二种情况的这个步骤给忽略了，主要是当时也不影响理解，现在既然遇到了，就讲一讲吧。

在设备生命线的过程中，设置完设备地址，让设备进入 `Address` 状态后，立马就调用了 `hub.c` 里一个名叫 `ep0_reinit` 的函数。

```
2066 static void ep0_reinit(struct usb_device *udev)
2067 {
2068     usb_disable_endpoint(udev, 0 + USB_DIR_IN);
2069     usb_disable_endpoint(udev, 0 + USB_DIR_OUT);
2070     udev->ep_in[0] = udev->ep_out[0] = &udev->ep0;
2071 }
```



这个函数中只对端点 0 调用了 `usb_disable_endpoint()`，但是端点 0 接下来还是要使用的，不然你就取不到设备那些描述符了，所以接着重新将 `ep0` 赋给 `ep_in[0]`和 `ep_out[0]`。多说无益，还是到 `usb_disable_endpoint()`里面去看看吧。

```
987 void usb_disable_endpoint(struct usb_device *dev, unsigned int epaddr)
988 {
989     unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
990     struct usb_host_endpoint *ep;
991
992     if (!dev)
993         return;
994
995     if (usb_endpoint_out(epaddr)) {
996         ep = dev->ep_out[epnum];
997         dev->ep_out[epnum] = NULL;
998     } else {
999         ep = dev->ep_in[epnum];
1000         dev->ep_in[epnum] = NULL;
1001     }
1002     if (ep && dev->bus)
1003         usb_hcd_endpoint_disable(dev, ep);
1004 }
```

这个函数先获得端点号和端点的方向，然后从 `ep_in` 或 `ep_out` 两个数组里取出端点的 `struct usb_host_endpoint` 结构体，并将数组里的对应项置为空，要注意的是这里同样不是释放掉数组里对应项的内存而是置为空。这两个数组里的 `ep_in[0]`和 `ep_out[0]`是早就被赋值了，至于剩下的那些项是在什么时候被赋值的，又是指向了什么东西，就是 `usb_set_configuration` 函数第三个阶段的事了。

最后 1003 行调用了一个 `usb_hcd_endpoint_disable` 函数，主要的工作还得它来做，不过这已经深入 HCD 的腹地了，就不多说了，还是回到 `usb_disable_device()`吧。在为每个端点都调用了 `usb_disable_endpoint()`之后，还有一个小步骤要做，就是将设备 `struct usb_device` 结构体的 `toggle` 数组置为 0。至于 `toggle` 数组有什么用，为什么要被初始化为 0，还是回首到设备那节去看吧。我要直接讲 `usb_set_configuration()`了。

1504 行，又一次与 `usb_control_msg()`相遇了，每当我们向设备发送请求时它就会适时出现。

`usb_control_msg` 这次出现的目的当然是为了 `SET_CONFIGURATION` 请求，这里只说一下它的那堆参数，看一下如图 1.33.1 所示的这张表。

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	配置值 Configuration Value	0	0	NULL

图 1.33.1 SET\_CONFIGURATION 请求

SET\_CONFIGURATION 请求不需要 DATA transaction，而且还是协议中规定所有设备都要支持的标准请求，也不是针对端点或者接口，而是针对设备的，所以 bRequestType 只能为 0x80，就是上面表里的 00000000B，也就是 1505 行的第一个 0。wValue 表示配置的 bConfigurationValue，就是 1505 行的 configuration。

1514 行，将激活的配置的地址赋给 actconfig。如果 cp 为空，重新设置设备的状态为 Address，并将之前准备的那些 struct usb\_interface 结构体和 new\_interfaces 释放掉，然后返回。看一下前面的代码，cp 有三种可能为空，一种是参数 configuration 为 -1，一种是参数 configuration 为 0，而且从设备的 config 数组里拿出来的就为空，还有一种是 SET\_CONFIGURATION 清除了问题。不管怎么说，到了 1515 行，cp 还是空的，你就要准备返回了。

1520 行，事情在这里发展达到了高潮的顶端，设置设备的状态为 configured。

## 34. 驱动的生命线（四）

设备自从有了 Address 拿到了各种描述符，就在那儿看 usb\_generic\_driver 忙活了，不过还算没白忙，设备总算是幸福的进入“configured”了。

Address 有点儿像你刚买的一套新房子，如果不装修，它对你来说的意义就是可以对人说你的家庭地址在哪儿了，可实际上你在里边儿什么也干不了，你还得想办法去装修它，configured 就像是已经装修好的房子，下面咱就看一看 usb\_generic\_driver 又对设备做了些什么。

1525 行，nintf 就是配置里接口的数目，那么这个 for 循环显然就是在对配置里的每个接口做处理。

1530 行，这里要明白的是，cp 里的两个数组 interface 和 intf\_cache，一个是没有初始化的，一个是已经动过手术很饱满的。正像前面提到的，这里需要为 interface 动手术，拿 intf\_cache 的内容去充实它。

1536 行，获得这个接口的 0 号设置。因为某些厂商有特殊的癖好，导致了 struct usb\_host\_config 结构中的数组 interface 并不一定是按照接口号的顺序存储的，必须使用 usb\_ifnum\_to\_if() 来获得指定接口号对应的 struct usb\_interface 结构体。现在咱们需要再多知道一点，接口里的 altsetting 数组也不一定是按照设置编号来顺序存储的，必须使用 usb\_altnum\_to\_altsetting() 来获得接口里的指定设置，它在 usb.c 中定义。

```
118 struct usb_host_interface *usb_altnum_to_altsetting(const struct
usb_interface *intf,
119 unsigned int altnum)
120 {
```

```

121     int i;
122
123     for (i = 0; i < intf->num_altsetting; i++) {
124         if (intf->altsetting[i].desc.bAlternateSetting == altnum)
125             return &intf->altsetting[i];
126     }
127     return NULL;
128 }

```

这个函数依靠一个 for 循环来解决问题，就是轮询接口里的每个设置，比较它们的编号与你指定的编号是不是一样。原理简单，操作也简单，不简单的是前面调用它时为什么指定编号 0，也就是获得 0 号设置。这还要归咎于在 spec 里，接口的默认设置总是 0 号设置，所以这里的目的就是获得接口的默认设置，如果没有拿到设置 0，接下来就在 1544 行拿 altsetting 数组里的第一项来充数。

1546 行，指定刚刚拿到的设置为当前要使用的设置。

1547 行，前面遇到过 device 和 endpoint 的 disable 函数，这里遇到个 interface 的 enable 函数，同样在 message.c 中定义。

```

1111 static void usb_enable_interface(struct usb_device *dev,
1112                                 struct usb_interface *intf)
1113 {
1114     struct usb_host_interface *alt = intf->cur_altsetting;
1115     int i;
1116
1117     for (i = 0; i < alt->desc.bNumEndpoints; ++i)
1118         usb_enable_endpoint(dev, &alt->endpoint[i]);
1119 }

```

这个函数同样也是靠一个 for 循环来解决问题，轮询前面获得的那个接口设置使用到的每个端点，调用 message.c 里的 usb\_enable\_endpoint() 将它们 enable。

```

1085 static void
1086 usb_enable_endpoint(struct usb_device *dev, struct usb_host_endpoint *ep)
1087 {
1088     unsigned int epaddr = ep->desc.bEndpointAddress;
1089     unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
1090     int is_control;
1091
1092     is_control = ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_
MASK)
1093                  == USB_ENDPOINT_XFER_CONTROL);
1094     if (usb_endpoint_out(epaddr) || is_control) {
1095         usb_settoggle(dev, epnum, 1, 0);
1096         dev->ep_out[epnum] = ep;
1097     }
1098     if (!usb_endpoint_out(epaddr) || is_control) {
1099         usb_settoggle(dev, epnum, 0, 0);
1100         dev->ep_in[epnum] = ep;
1101     }
1102 }

```

这个函数的前面几行没什么好说的，分别获得端点地址，端点号，还有是不是控制端点。后面的两个 if 函数，它们分别根据端点的方向来初始化设备的 `ep_in` 和 `ep_out` 数组，还有就是调用了 `include/linux/usb.h` 里的一个叫 `usb_settoggle` 的宏。

```
1425 #define usb_gettoggle(dev, ep, out) (((dev)->toggle[out] >> (ep)) & 1)
1426 #define usb_dotoggle(dev, ep, out) ((dev)->toggle[out] ^= (1 << (ep)))
1427 #define usb_settoggle(dev, ep, out, bit) \
1428     ((dev)->toggle[out] = ((dev)->toggle[out] & ~(1 << (ep))) | \
1429     ((bit) << (ep)))
```

这三个宏都是用来处理端点的 `toggle` 位的，也就是 `struct usb_device` 里的数组 `toggle[2]`。`toggle[0]`对应的是 IN 端点，`toggle[1]`对应的是 OUT 端点，上面宏参数中的 `out` 用来指定端点是 IN 还是 OUT，`ep` 指的并不是端点的结构体，而仅仅是端点号。

`usb_gettoggle` 用来得到端点对应的 `toggle` 值，`usb_dotoggle` 用来将端点的 `toggle` 位取反，也就是原来为 1 就置为 0，原来为 0 就置为 1，`usb_settoggle` 看起来就要复杂点儿，意思是如果 `bit` 为 0，则将 `ep` 所对应的 `toggle` 位 `reset` 成 0，如果 `bit` 为 1，则 `reset` 为 1。`((dev)->toggle[out] & ~(1<<(ep)))`就是把 1 左移 `ep` 位，比如 `ep` 为 3，那么就是得到了 1000，然后取反，得到 0111，（当然高位还有更多个 1），然后 `(dev)->toggle[out]`和 0111 相与，这就是使得 `toggle[out]`的第 3 位清零而其他位都不变。这里调用 `usb_settoggle` 时，`bit` 传递的是 0，用来将端点的 `toggle` 位清零，原因就是对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为 `DATA0` 的，然后才一次传 `DATA0`，一次传 `DATA1`。

现在看一看为什么两个 if 语句里都出现有 `is_control`。控制传输使用的是 `message` 管道，`message` 管道必须对应两个相同号码的端点，一个用来“in”，一个用来“out”。这里使用两个 if，而不是 if-else 组合，目的就是加进去一个 `is_control`，表示只要是控制端点，就将它的端点号对应的 IN 和 OUT 两个方向上的 `toggle` 位还有 `ep_int` 和 `ep_out` 都给初始化。当然，所谓的控制端点一般也就是指端点 0。

`endpoint` 的 `enable` 函数要比 `disable` 函数简单得多，`disable` 时还要深入到 `HCD` 的腹地去撤销挂在它上面的各个 `urb`，而 `enable` 时就是简单设置一下 `toggle` 位还有那两个数组就好了，要知道它的 `urb` 队列 `urb_list` 是早在从设备那里获取配置描述符并去解析那一大堆数据时就初始化好了的。`enable` 之后，接口里的各个端点便都处于了可用状态，你就可以在驱动里向指定的端点提交 `urb` 了。当然，目前这个时候接口还仍然是接口，驱动（接口驱动）还仍然是驱动。

然后看一看跟在 `usb_enable_interface()` 后面的那几行，接口所属的总线类型仍然为 `usb_bus_type`，设备类型变为 `usb_if_device_type`，`dma_mask` 被设置为你的设备的 `dma_mask`，而你设备的 `dma_mask` 很早以前就被设置为了 `host controller` 的 `dma_mask`。

1553 行，`device_initialize` 在初始化设备 `struct usb_device` 结构体时遇到过一次，这里初始化接口时又遇到了。

1554 行，将接口的 `is_active` 标志初始化为 0，表示它还没有与任何驱动绑定，就是还没有找到另一半。

1559 行，`for` 循环结束了，`new_interfaces` 的历史使命也就结束了。这里的 `kfree` 释放的只是 `new_interfaces` 指针数组的内存，而不包括它里面各个指针所指向的内存，至于那些数据，都已经在前面被赋给配置里的 `interface` 数组了。

1561 行，获得配置的字符串描述符。

1570 行，这个 `for` 循环结束，`usb_set_configuration()` 的三个阶段也就算结束了，设备和 `usb_generic_driver` 上上下下忙活了这么久也都很累了，接下来就该接口和接口驱动去忙活了。

这个 `for` 循环将前面那个 `for` 循环准备好的每个接口送给设备模型，Linux 设备模型会根据总线类型 `usb_bus_type` 将接口添加到 USB 总线的那条有名的设备链表里，然后去轮询 USB 总线的另外一条有名的驱动链表，针对每个找到的驱动去调用 USB 总线的 `match` 函数，也就是 `usb_device_match()`，去为接口寻找另一个匹配的半圆。你说这个时候设备和接口两条路它应该走哪条？它的类型已经设置成 `usb_if_device_type` 了，设备那条路“看门儿”的根本就不会让它进，所以它必须得去走接口那条路。

---

## 35. 字符串描述符

字符串描述符的地位仅次于设备/配置/接口/端点四大描述符，那么四大设备必须得支持它。而字符串描述符对设备来说则是可选的。

这并不是说字符串描述符不重要，对咱们来说，字符串要比数字亲切得多，提供字符串描述符的设备也要比没有提供的设备亲切得多，不会有人专门去记前面使用 `lsusb` 列出的 `04b4` 表示 Cypress Semiconductor Corp.。

一提到字符串，不可避免就得提到字符串使用的语言。字符串亲切是亲切，但不像数字那样全球通用。当然这并不是说设备中就要存储这么多不同语言的字符串描述符，这未免要求过高了一些，代价也昂贵了一些，要知道这些描述符不会凭空生出来，是要存储在设备的 EEPROM 里的，此物是需要记忆的。所以说只提供几种语言的字符串描述符就可以了，甚至说只提供一种语言，比如英语就可以了。

其实咱们现在说的语言就是太多了，不过不管哪种语言，在 PC 里或者设备中存放都只能用二进制数字，这就需要在语言与数字之间进行编码，这个所谓的编码和这个世界上其他事物一样，都是有多种的。

Spec 里就说了，字符串描述符使用的就是 UNICODE 编码，USB 设备中的字符串可以通过它来支持多种语言，不过你需要在向设备请求字符串描述符时指定一个你期望看到的一种语言，俗称语言 ID，即 Language ID。这个语言 ID 使用两个字节表示，所有可以使用的语言 ID 在 [http://www.usb.org/developers/docs/USB\\_LANGIDs.pdf](http://www.usb.org/developers/docs/USB_LANGIDs.pdf) 文档里都有列出来，从这个文档里你也可以明白为什么要使用两个字节，而不是一个字节表示。

这么说吧，比如中文是 0X04，但是中文还有好几种，所以需要另外一个字节表示是哪一种中文，简体就是 0X02。注意合起来表示简体中文并不是 0X0402 或者 0X0204，因为这两个字节并不是分得那么清，bit0~bit 9 一共 10 位去表示 Primary 语言 ID，其余 6 位去表示 Sub 语言 ID，毕竟一种语言的 Sub 语言不可能特别多，没必要分去整整一半 8bits，所以简体中文的语言 ID 就是 0X0804。

不多啰嗦，还是结合代码从 `usb_cache_string` 说起，看一看如何去获得一个字符串描述符，它在 `message.c` 中定义。

```

825 char *usb_cache_string(struct usb_device *udev, int index)
826 {
827     char *buf;
828     char *smallbuf = NULL;
829     int len;
830
831     if (index > 0 && (buf = kmalloc(256, GFP_KERNEL)) != NULL) {
832         if ((len = usb_string(udev, index, buf, 256)) > 0) {
833             if ((smallbuf = kmalloc(++len, GFP_KERNEL)) == NULL)
834                 return buf;
835             memcpy(smallbuf, buf, len);
836         }
837         kfree(buf);
838     }
839     return smallbuf;
840 }
```

每个字符串描述符都有一个序号，字符串描述符序号是不能重复的。

字符串描述符当然可以有很多位，参数 `index` 就是表示了你希望获得其中的第几个。但是不可疏忽大意的是，不能指定 `index` 为 0，0 编号是有特殊用途的，指定为 0 就什么也得不到。

有关 `usb_cache_string()` 函数，还需要明白两点，第一是它采用的方针策略，就是苦活儿累活儿找 `usb_string()` 去做。这个 `usb_string()` 怎么工作的之后再看，现在只要注意一下它的参数，比 `usb_cache_string()` 的参数多了两个，`buf` 和 `size`，也就是需要传递一个存放返回的字符串描述符的缓冲区。但是你调用 `usb_cache_string()` 时并没有指定一个明确的 `size`，`usb_cache_string()` 也就不知道你想要的字符串描述符有多大，于是它就采用了这么一个技巧，先申请一个足够大的缓冲区（这里是 256 字节），拿这个缓冲区去调用 `usb_string()`，通过 `usb_string()` 的返回值会得到字符串描述符的真实大小，然后再拿这个值去申请一个缓冲区，并将大缓冲区里放的字符串描述符数据复制过来，这时那个大缓冲区当然就没什么利用价值了，于是再把它给释放掉。

第二就是 `usb_cache_string()` 申请小缓冲区时，使用的并不是 `usb_string()` 的返回值，而是多了 1 个字节，也就是说要从大缓冲区里多复制 1 个字节到小缓冲区里，为什么？这牵涉到 C 语言里字符串方面字符串结束符。

字符串都需要结束符，但并不是每个人都能记得给字符串加上结束符。下面举一个例子。

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[])
5 {
6     #define MAX (100)
7
8     char buf[512], tmp[32];
9     int i, count = 0;
10
11     for ( i = 0; i< MAX; ++i){
12         sprintf(tmp, "0x%.4X", i);
13
14         strcat(buf, tmp);
15
16         if (count++ == 10){
17             printf("%s\n", buf);
18             buf[0] = '\0';
19             count = 0;
20         }
21     }
22
23     return 0;
24 }
```

这程序简单直白：打印 100 个数，每行 10 个。你只需要查看它能不能得到预期的结果。

当然这程序是有问题的，在第 10 行少了下面这一句：

```
buf[0] = '\0'; //////////////// here !
```

也就是说忘记将 `buf` 初始化了，传递给 `strcat` 的是一个没有初始化的 `buf`，这个时候 `buf` 的内容都非 0，`strcat` 不知道它的结尾在哪里，它能猜到 `buf` 的开始，却猜不到 `buf` 的结束。memset 就比较的消耗 CPU 了，而这里 `buf[0] = '\0'` 就足够用了。为了更好地说明问题，这里放上内核中对 `strcat` 的定义。

```
171 char *strcat(char *dest, const char *src)
172 {
173     char *tmp = dest;
174
175     while (*dest)
176         dest++;
177     while ((*dest++ = *src++) != '\0')
178         ;
179     return tmp;
180 }
```

`strcat` 会从 `dest` 的起始位置开始去寻找一个字符串的结束符，只有找到 175 行的 `while` 循环才会结束。但是如果 `dest` 没有初始化过，义无反顾地“`strcat`”并不会会有好结果。本来 `strcat` 的目的是将 `tmp` 追加到 `buf` 里字符串的后面，但是因为 `buf` 没有初始化，没有一个结束符，`strcat` 就会一直找下去，就算它在哪儿停了下来，如果这个停下的位置超出了 `buf` 的范围，就会把 `src` 的数据写到不应该写的地方，就可能会破坏其他很重要的数据，你的系统可能就“死”掉了。

解决这种问题的方法很简单，就是记着在使用指针、数组前首先统统初始化，到最后觉得哪里影响 CPU 性能了再去优化它。

问一个问题，这个字符串结束符具体是什么？C 和 C++ 里一般是指 `'\0'`，像上面的那句 `buf[0] = '\0'`。这里再引用 `spec` 里的一句话：“The UNICODE string descriptor is not NULL-terminated.” 什么是 NULL-terminated 字符串？其实就是以 `'\0'` 结尾的字符串，下面查看一下内核中对 NULL 的定义。

```
6 #undef NULL
7 #if defined(__cplusplus)
8 #define NULL 0
9 #else
10 #define NULL ((void *)0)
11 #endif
```

0 是一个整数，但它又不仅仅是一个整数。由于各种标准转换，0 可以被用于作为任意的整型、浮点类型、指针。0 的类型将由上下文来确定。典型情况下，0 被表示为一个适当大小的全零二进制的模式。所以，无论 NULL 是定义为常数 0 还是 `((void *)0)` 这个零指针，NULL 都是指的是 0 值，而不是非 0 值。而字符的 `'\0'` 和整数的 0 也可以通过转型来相互转换。再多说一点，`'\0'` 是 C 语言定义的用 `'\'` 加八进制 ASCII 码来表示字符的一种方法，`'\0'` 就是表示一个 ASCII 码值为 0 的字符。

所以更准确地说，NULL-terminated 字符串就是以 0 值结束的字符串，那么 `spec` 的那句话说字符串描述符不是一个 NULL-terminated 字符串，意思也就是字符串描述符没有一个结束符，你从设备那里得到字符串之后得给它追加一个结束符。本来 `usb_string()` 里已经为 `buf` 追加好了，但是它返回的长度里还是没有包括这个结束符的 1 个字节，所以 `usb_cache_string()` 为 `smallbuf` 申请内存时就得多准备那么一个字节，以便将 `buf` 里的那个结束符也给复制过来。现在就查看 `usb_string()` 的细节，定义在 `message.c` 里。

```
757 int usb_string(struct usb_device *dev, int index, char *buf, size_t size)
758 {
759     unsigned char *tbuf;
760     int err;
761     unsigned int u, idx;
762
763     if (dev->state == USB_STATE_SUSPENDED)
764         return -EHOSTUNREACH;
765     if (size <= 0 || !buf || !index)
766         return -EINVAL;
767     buf[0] = 0;
```



```

768     tbuf = kmalloc(256, GFP_KERNEL);
769     if (!tbuf)
770         return -ENOMEM;
771
772     /* get langid for strings if it's not yet known */
773     if (!dev->have_langid) {
774         err = usb_string_sub(dev, 0, 0, tbuf);
775         if (err < 0) {
776             dev_err (&dev->dev,
777                     "string descriptor 0 read error: %d\n",
778                     err);
779             goto errout;
780         } else if (err < 4) {
781             dev_err (&dev->dev, "string descriptor 0 too short\n");
782             err = -EINVAL;
783             goto errout;
784         } else {
785             dev->have_langid = 1;
786             dev->string_langid = tbuf[2] | (tbuf[3]<< 8);
787             /* always use the first langid listed */
788             dev_dbg (&dev->dev, "default language 0x%04x\n",
789                     dev->string_langid);
790         }
791     }
792
793     err = usb_string_sub(dev, dev->string_langid, index, tbuf);
794     if (err < 0)
795         goto errout;
796
797     size--; /* leave room for trailing NULL char in output buffer */
798     for (idx = 0, u = 2; u < err; u += 2) {
799         if (idx >= size)
800             break;
801         if (tbuf[u+1]) /* high Byte */
802             buf[idx++] = '?'; /* non ISO-8859-1 character */
803         else
804             buf[idx++] = tbuf[u];
805     }
806     buf[idx] = 0;
807     err = idx;
808
809     if (tbuf[1] != USB_DT_STRING)
810         dev_dbg(&dev->dev, "wrong descriptor type %02x for string %d
811 (%s)\n", tbuf[1], index, buf);
812     errout:
813     kfree(tbuf);
814     return err;
815 }

```

763 行，这几行做些例行检查，设备不能是挂起的，index 也不能是 0，只要传递了指针就需要检查。

767 行，初始化 buf，usb\_cache\_string()并没有对这个 buf 初始化，所以这里必须要加上这么一步。当然 usb\_string()并不仅仅只有在 usb\_cache\_string()里调用，可能会在很多地方调用到它，不过不管在哪里，这里为了谨慎起见，还是需要这一步。

768 行，申请一个 256 字节大小的缓冲区。前面一直强调要初始化，怎么到这里没有去初始化 tbuf？这是因为没必要。为什么没必要？查看一下 usb\_string()最后面的那些代码就明白了。

773 行，struct usb\_device 里有 have\_langid 和 string\_langid 这么两个字段是和字符串描述符有关的，string\_langid 用来指定使用哪种语言，have\_langid 用来指定 string\_langid 是否有效。如果 have\_langid 为空，就说明没有指定使用哪种语言，那么获得的字符串描述符使用的是哪种语言就完全看设备了。你可能会疑惑，为什么当 have\_langid 为空时，要在 774 行和 793 行调用两次 usb\_string\_sub()？就像 usb\_string()是替 usb\_cache\_string()做“苦工”一样，usb\_string\_sub()是替 usb\_string()做“苦工”的，也就是说 usb\_string()是靠 usb\_string\_sub()去获得字符串描述符的，那为什么 have\_langid 为空时，要获取两遍的字符串描述符？

你可以比较一下两次调用 usb\_string\_sub()的参数有什么区别。第一次调用参数时，语言 ID 和 index 都为 0，第二次调用参数时就明确指定了语言 ID 和 index。这里的玄机就在 index 为 0 时，也就是 0 编号的字符串描述符是什么，前面只说了它有特殊的用途，现在必须得解释一下。

spec 在 9.6.7 节说了，编号 0 的字符串描述符包括了设备所支持的所有语言 ID，对应的就是如图 1.35.1 所示的表。

偏移(Offset)	字段(Field)	字节(Size)	描述(Description)
0	bLength	1	描述符的字节数
1	bDescriptorType	1	描述符的类型
2	wLANGID[0]	2	语言 ID 0
...	...	...	...
N	wLANGID[x]	2	语言 ID x

图 1.35.1 0 号字符串描述符

第一次调用 usb\_string\_sub()就是为了获得这张表，获得这张表做什么用？接着往下看。

775 行，usb\_string\_sub()返回一个负数，就表示没有获得这张表，没有取到 0 号字符串描述符。如果返回值比 4 要小，就表示获得的表里没有包含任何一个语言 ID。要知道一个语言 ID 占用 2 个字节，还有前两个字节表示表的长度及类型，所以得到的数据至少要为 4，才能够得到一个有效的语言 ID。如果返回值比 4 要大，就使用获得的数据的第 3 字节和第 4 字节设置 string\_langid，同时设置 have\_langid 为 1。

现在很明显了，773 行到 791 行这些代码就是在你没有指定使用哪种语言时，去获取设备中默认使用的语言，也就是 0 号字符串描述符里的第一个语言 ID 所指定的语言。如果没有找到这个默认的语言 ID，即 usb\_string\_sub()返回值小于 4 的情况，就没有办法再去获得其他字符串描述符了。因为没有指定语言，设备不知道你是要求的是英语还是中文或是其他的。

793 行，使用指定的语言 ID，或者前面获得的默认语言 ID 去获得想要的字符串描述符。现在看一看定义在 message.c 里的 usb\_string\_sub 函数。

```

696 static int usb_string_sub(struct usb_device *dev, unsigned int langid,
697                          unsigned int index, unsigned char *buf)
698 {
699     int rc;
700
701     /* Try to read the string descriptor by asking for the maximum
702      * possible number of Bytes */
703     if (dev->quirks & USB_QUIRK_STRING_FETCH_255)
704         rc = -EIO;
705     else
706         rc = usb_get_string(dev, langid, index, buf, 255);
707
708     /* If that failed try to read the descriptor length, then
709      * ask for just that many Bytes */
710     if (rc < 2) {
711         rc = usb_get_string(dev, langid, index, buf, 2);
712         if (rc == 2)
713             rc = usb_get_string(dev, langid, index, buf, buf[0]);
714     }
715
716     if (rc >= 2) {
717         if (!buf[0] && !buf[1])
718             usb_try_string_workarounds(buf, &rc);
719
720         /* There might be extra junk at the end of the descriptor */
721         if (buf[0] < rc)
722             rc = buf[0];
723
724         rc = rc - (rc & 1); /* force a multiple of two */
725     }
726
727     if (rc < 2)
728         rc = (rc < 0 ? rc : -EINVAL);
729
730     return rc;
731 }

```

这个函数首先检查一下你的设备是不是属于合格的设备，然后就调用 usb\_get\_string() 去获得字符串描述符。USB\_QUIRK\_STRING\_FETCH\_255 就是在 include/linux/usb/quirks.h 中定义的那些形形色色的“毛病”之一，表示设备在获取字符串描述符时会“crash”。

usb\_string\_sub() 的核心就是 message.c 中定义 usb\_get\_string 函数。

```

664 static int usb_get_string(struct usb_device *dev, unsigned short langid,
665                          unsigned char index, void *buf, int size)
666 {
667     int i;
668     int result;
669
670     for (i = 0; i < 3; ++i) {
671         /* retry on length 0 or stall; some devices are flakey */
672         result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
673                                USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,

```

```
674         (USB_DT_STRING << 8) + index, langid, buf, size,
675         USB_CTRL_GET_TIMEOUT);
676         if (!(result == 0 || result == -EPIPE))
677             break;
678     }
679     return result;
680 }
```

我已经不记得这是第几次遇到 `usb_control_msg()` 了。还是简单说一下它的参数。`wValue` 的高位字节表示描述符的类型, 低位字节表示描述符的序号, 所以有 674 行的 `(USB_DT_STRING << 8) + index`, `wIndex`。对于字符串描述符应该设置为使用语言的 ID, 所以有 674 行的 `langid`。至于 `wLength`, 就是描述符的长度, 对于字符串描述符很难有一个统一的确定长度, 所以一般来说上面传递过来的通常是一个比较大的 255 字节。

和获得设备描述符时一样, 因为一些厂商生产出的设备“古灵精怪”, 可能需要多试几次才能成功。要容许设备犯错误。

还是回过头去看 `usb_string_sub` 函数, 如果 `usb_get_string()` 成功地得到了期待的字符串描述符, 则返回获得的字节数, 如果这个数目小于 2, 就再读两个字节试一试, 要想明白这两个字节是什么内容, 需要查看如图 1.35.2 所示的表。

偏移 (Offset)	字段 (Field)	字节 (Size)	描述 (Description)
0	bLength	1	描述符的字节数
1	bDescriptorType	1	描述符的类型
2	bString	N	UNICODE 编码的字符串

图 1.35.2 其他字符串描述符

图 1.35.1 描述的是 0 号字符串描述符的格式, 图 1.35.2 描述的是其他字符串描述符的格式。很明显可以看到, 它的前两个字节分别表示了长度和类型, 如果读两个字节成功的话, 就可以准确地获得这个字符串描述符的长度, 然后再拿这个准确的长度去请求一次。

该尝试的都尝试了, 现在看 716 行, 分析一下前面调用 `usb_get_string()` 的结果, 如果几次尝试之后, 它的返回值还是小于 2, 那就返回一个错误码。`rc` 大于等于 2, 说明终于获得了一个有效的字符串描述符。

717 行, `buf` 的前两个字节有一个为空时, 也就是字符串描述符的前两个字节有一个为空时, 调用了 `message.c` 中定义的 `usb_try_string_workarounds` 函数。

```
682 static void usb_try_string_workarounds(unsigned char *buf, int *length)
683 {
684     int newlength, oldlength = *length;
```

```

685
686     for (newlength = 2; newlength + 1 < oldlength; newlength += 2)
687         if (!isprint(buf[newlength]) || buf[newlength + 1])
688             break;
689
690     if (newlength > 2) {
691         buf[0] = newlength;
692         *length = newlength;
693     }
694 }

```

这个函数的目的是从 `usb_get_string()` 返回的数据里计算出前面有效的长度。它的核心就是 686 行的 `for` 循环，不过要搞清楚这个循环，还真不是一件容易的事情，得有相当的理论功底。

前面刚说了字符串描述符使用的是 `UNICODE` 编码，其实 `UNICODE` 指的是包含了字符集、编码、字型等很多规范的一整套系统，字符集仅仅描述符系统中存在那些字符，并进行分类，并不涉及如何用数字编码表示的问题。

`UNICODE` 使用的编码形式主要有两种 `UTF`，即 `UTF-8` 和 `UTF-16`。使用 `usb_get_string()` 获得的字符串使用的是 `UTF-16` 编码规则，而且是 `little-endpoint`，每个字符都需要使用两个字节来表示。在这个 `for` 循环里 `newlength` 每次加 2，就是表示每次处理一个字符，但是要弄明白怎么处理的，还需要知道这两个字节分别是什么，这就不得不提及 `ASCII`、`ISO-8859-1` 等几个名词。

`ASCII` 是用来表示英文的一种编码规范，表示的最大字符数为 256，每个字符占 1 个字节。但是英文字符没这么多，一般来说 128 个也就够了（最高位为 0），这就已经完全包括了控制字符、数字、大小写字母，还有其他一些符号。对于法语、西班牙语和德语之类的西欧语言都使用叫做 `ISO-8859-1`，它扩展了 `ASCII` 码的最高位，来表示 Ñ（241），和 Ü（252）这样的字符。而 `Unicode` 的低字节，也就是在 0 到 255 同 `ISO-8859-1` 完全一样，它接着使用剩余的数字，256 到 65535，扩展到表示其他语言的字符。可以说 `ISO-8859-1` 就是 `Unicode` 的子集，如果 `Unicode` 的高字节为 0，则它表示的字符就和 `ISO-8859-1` 完全一样了。

再看一看 686 行这个 `for` 循环，`newlength` 从 2 开始，是因为前两个字节应该是表示长度和类型的，这里只逐个儿对上上面 Table 9-16 里的 `bString` 中的每个字符做处理。还要知道 `usb_get_string()` 得到的结果是 `little-endpoint` 的，所以 `buf[newlength]` 和 `buf[newlength + 1]` 分别表示一个字符的低字节和高字节，那么 `isprint(buf[newlength])` 就是用来判断一下这个 `Unicode` 字符的低字节是不是可以 `print` 的。如果不是，就没必要再往下循环了，后边儿的字符也不再看了，然后就到了 690 行的 `if` 函数，将 `newlength` 赋给 `buf[0]`，即 `bLength.length` 指向的是 `usb_get_string()` 返回的原始数据的长度，692 行使用 `for` 循环计算出的有效长度将它修改了。`isprint` 在 `include/linux/ctype.h` 中定义。

这个 `for` 循环终止的条件有两个，另外一个就是 `buf[newlength + 1]`，也就是这个 `Unicode` 字符的高字节不为 0，这时它不存在对应的 `ISO-8859-1` 形式，为什么加上这个判断？接着往下看。

`usb_string_sub()` 的 721 行, `buf[0]` 表示的就是 `bLength` 的值, 如果它小于 `usb_get_string()` 获得的数据长度, 说明这些数据里存在一些垃圾, 要把它们给揪出来排除掉。要知道这个 `rc` 是要作为真实有效的描述符长度返回的, 所以这个时候需要将 `buf[0]` 赋给 `rc`。

724 行, 每个 Unicode 字符需要使用两个字节来表示, 所以 `rc` 必须为偶数, 即 2 的整数倍。如果为奇数, 就得将最后那一个字节给去掉, 也就是将 `rc` 减 1。我们可以学习一下这里将一个数字转换为偶数时采用的技巧,  $(rc \& 1)$  在 `rc` 为偶数时等于 0, 为奇数时等于 1, 再使用 `rc` 减去它, 得到的就是一个偶数。

从 716 行到 725 行这几行, 应该可以看出, 在成功获得一个字符串描述符时, `usb_string_sub()` 返回的是一个 NULL-terminated 字符串的长度, 并没有涉及结束符。牢记这一点, 回到 `usb_string` 函数的 797 行, 先将 `size`, 也就是 `buf` 的大小减 1, 目的就是为结束符保留 1 个字节的位置。

798 行, `tbuf` 里保存的是 `GET_DESCRIPTOR` 请求返回的原始数据, `err` 是 `usb_string_sub()` 的返回值, 一切顺利的话, 它表示的就是一个有效的描述符的大小。这里 `idx` 的初始值为 0, 而 `u` 的初始值为 2, 目的是从 `tbuf` 的第三个字节开始复制给 `buf`, 毕竟这里的目的是获得字符串描述符里的 `bString`, 而不是整个描述符的内容。`u` 每次都是增加 2, 这是因为采用的 UTF-16 是用两个字节表示一个字符的, 所以循环一次要处理两个字节。

801 行, 这个 `if-else` 组合你可能比较糊涂, 要搞清楚, 还要看一下前面刚讲过的一些理论。`tbuf` 里每个 Unicode 字符两个字节, 又是 little-endpoint 的, 所以 801 行就是判断这个 Unicode 字符的高位字节是不是为 0, 如果不为 0, 则 ISO-8859-1 里没有这个字符, 就设置 `buf` 里的对应字符为 '?'. 如果它的高位字节为 0, 就说明这个 Unicode 字符 ISO-8859-1 里也有, 就直接将它的低位字节赋给 `buf`。一个 `for` 循环下来, 就将 `tbuf` 里的 Unicode 字符串转化成了 `buf` 里的 ISO-8859-1 字符串。

806 行, 为 `buf` 追加一个结束符。

## 36. 接口的驱动

我们已经知道, `usb_generic_driver` 在自己的生命线里, 以一己之力将设备的各个接口送给了 Linux 的设备模型, 让 USB 总线的 `match` 函数, 也就是 `usb_device_match()`, 在自己的驱动链表里为它们寻找一个合适的接口驱动程序。现在让我问一句: “这些接口驱动都从哪里来?”

这就要说到几个著名的命令 `insmod`, `modprobe`, `rmmod`。当 `insmod` 或 `modprobe` 驱动时, 经过一个曲折的过程, 会调用到你驱动里的 `xxx_init` 函数, 进而去调用 `usb_register()` 将你的驱动提交给设备模型, 添加到 USB 总线的驱动链表里。`rmmod` 驱动时候, 同样经过一个曲折的过

程之后，调用到驱动里的 `xxx_cleanup` 函数，进而调用 `usb_deregister()` 将你的驱动从 USB 总线的驱动链表里删除掉。

现在就查看 `include/linux/usb.h` 中定义的 `usb_register` 函数。

```
916 static inline int usb_register(struct usb_driver *driver)
917 {
918     return usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME);
919 }
```

看到这个函数，让人不得不感叹一下，现在什么都要讲究包装，在内核中注册一个驱动也要包装两个层。

本来在两年以前没有 `usb_register_driver` 这个函数，只有个 `usb_register()`。而且那个时候 `struct usb_driver` 结构中还有一个有名的 `owner` 字段，每个在那个岁月里写过 USB 驱动的人都会认得它，并且都会毫不犹豫地将它设置为 `THIS_MODULE`。但是经过岁月的洗礼，在早先贴出来的 `struct usb_driver` 结构内容里，你会发现 `owner` 已经无影无踪了。要搞清楚这个历史变迁的来龙去脉，你得知道这个 `owner` 和 `THIS_MODULE` 都代表了什么。

从那个时代走过来的人，都应该知道 `owner` 是一个 `struct module *` 类型的结构体指针，现在告诉你的是每个 `struct module` 结构体在内核中都代表了一个内核模块，每个 `struct module` 结构体代表了什么模块，对它进行初始化的模块才知道。

当然，初始化这个结构不是写驱动的人该做的事，是在刚才那个从 `insmod` 或 `modprobe` 到你驱动的 `xxx_init` 函数的曲折过程中做的事。`insmod` 命令执行后，会在 `kernel/module.c` 里的一个系统调用 `sys_init_module`，它会调用 `load_module` 函数，将用户空间传入的整个内核模块文件创建成一个内核模块，并返回一个 `struct module` 结构体，从此，内核中便以这个结构体代表这个内核模块。

再查看 `THIS_MODULE` 宏是什么意思，它在 `include/linux/module.h` 里的定义：

```
85 #define THIS_MODULE (&__this_module)
```

它是一个 `struct module` 变量，代表当前模块，与著名的 `current` 有几分相似，可以通过 `THIS_MODULE` 宏来引用模块的 `struct module` 结构。比如使用 `THIS_MODULE->state` 可以获得当前模块的状态。现在你应该明白为什么在以前，你需要毫不犹豫地将 `struct usb_driver` 结构中的 `owner` 设置为 `THIS_MODULE`，这个 `owner` 指针指向的就是模块自己。

那现在 `owner` 怎么就没了呢？这个说来可就话长了，就长话短说吧。不知道你有没有忘记过初始化 `owner`，反正是很多人都会忘记，大家都把注意力集中到 `probe`、`disconnect` 等变量上面了，这个不需要动脑子，只需要花个几秒钟指定一下的 `owner` 的操作反倒常常被忽视。

于是在 2006 年的春节前夕，Greg 去掉了 `owner`，于是千千万万个写 USB 驱动的人再也不用去时刻谨记初始化 `owner` 了。我们是不用设置 `owner` 了，可 Core 里不能不设置，`struct usb_driver`

结构中不是没有 owner 了吗,可它里面嵌的 struct device\_driver 结构中还有,设置了它就可以了。于是 Greg 同时又增加了 usb\_register\_driver()这么一层,usb\_register()可以通过将参数指定为 THIS\_MODULE 去调用它,所有的事情都挪到里面去做。反正 usb\_register()也是内联的,并不会增加调用的开销。现在是时机看一看 usb\_register\_driver 函数了。

```

734 int usb_register_driver(struct usb_driver *new_driver, struct module *owner,
735                        const char *mod_name)
736 {
737     int retval = 0;
738
739     if (usb_disabled())
740         return -ENODEV;
741
742     new_driver->drvwrap.for_devices = 0;
743     new_driver->drvwrap.driver.name = (char *) new_driver->name;
744     new_driver->drvwrap.driver.bus = &usb_bus_type;
745     new_driver->drvwrap.driver.probe = usb_probe_interface;
746     new_driver->drvwrap.driver.remove = usb_unbind_interface;
747     new_driver->drvwrap.driver.owner = owner;
748     new_driver->drvwrap.driver.mod_name = mod_name;
749     spin_lock_init(&new_driver->dynids.lock);
750     INIT_LIST_HEAD(&new_driver->dynids.list);
751
752     retval = driver_register(&new_driver->drvwrap.driver);
753
754     if (!retval) {
755         pr_info("%s: registered new interface driver %s\n",
756               usbcore_name, new_driver->name);
757         usbfs_update_special();
758         usb_create_newid_file(new_driver);
759     } else {
760         printk(KERN_ERR "%s: error %d registering interface "
761               "      driver %s\n",
762               usbcore_name, retval, new_driver->name);
763     }
764
765     return retval;
766 }

```

这函数和前面见过的 usb\_register\_device\_driver 长得很像,你如果是从那里一路看过来的话,不用我说什么,你都会明明白白它的意思。for\_devices 在 742 行设置成了 0,有了这行,match 里的 is\_usb\_device\_driver “把门儿”的才不会把它当成设备驱动放过去。

然后就是在 752 行将你的驱动提交给设备模型,从而添加到 USB 总线的驱动链表里,从此之后,接口和接口驱动就可以通过 USB 总线的 match 函数传达数据了。



## 37. 还是那个 match

从前面遇到 USB 总线的 match 函数 `usb_device_match()` 开始到现在，遇到了设备，遇到了设备驱动，遇到了接口，也遇到了接口驱动，期间还多次遇到 `usb_device_match()`。

设备没有真正自由过，刚开始时在 Default 状态动弹不得，稍后步入 Address，无论外头风光多好，都得与 `usb_generic_driver` 长相厮守，没得选择，终于达到了 Configured，又必须为自己的接口殚精竭虑，以便 `usb_device_match()` 能够为它们找一个好人家。

不管怎么说，在这里我们会再次与 `usb_device_match()` 相遇，查看它怎么在接口和驱动之间搭起那座桥。

```

540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545         /* interface drivers never match devices */
546         if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551
552     } else {
553         struct usb_interface *intf;
554         struct usb_driver *usb_drv;
555         const struct usb_device_id *id;
556
557         /* device drivers never match interfaces */
558         if (is_usb_device_driver(drv))
559             return 0;
560
561         intf = to_usb_interface(dev);
562         usb_drv = to_usb_driver(drv);
563
564         id = usb_match_id(intf, usb_drv->id_table);
565         if (id)
566             return 1;
567
568         id = usb_match_dynamic_id(intf, usb_drv);
569         if (id)
570             return 1;
571     }
572
573     return 0;
574 }
```

设备那条路已经走过了，现在走一走 552 行接口这条路。558 行，接口驱动的 `for_devices` 在 `usb_register_driver()` 里被初始化为 0，所以这个“把门儿”的会痛痛快快地放行，继续往下走。

561 行，遇到一对似曾相识的宏 `to_usb_interface` 和 `to_usb_driver`，之所以说似曾相识，是因为前面已经遇到过一对 `to_usb_device` 和 `to_usb_device_driver`。这两对宏一对用于接口和接口驱动，一对用于设备和设备驱动，意思都很直白，还是看一看 `include/linux/usb.h` 里的定义：

```
159 #define to_usb_interface(d) container_of(d, struct usb_interface, dev)
857 #define to_usb_driver(d) container_of(d, struct usb_driver, drvwrap.driver)
```

再往下走，就是两个函数 `usb_match_id` 和 `usb_match_dynamic_id`，它们都是用来完成实际的匹配工作的，只不过前一个函数是从驱动表的 `id_table` 里找，看接口是不是被驱动所支持，后一个函数是在驱动表的动态 `id` 链表 `dynids` 里找。驱动表的 `id` 表分 `id_table` 和 `dynids` 两种。显然 564 行到 570 行这几行的意思就是将 `id_table` 放在一个比较高的优先级的位置，从它里面找不到接口了才再从动态 `id` 链表里找。

当时讲到 `struct usb_driver` 结构时并没有详细讲它里面表示动态 `id` 的结构体 `struct usb_dynids`，所以现在补充一下，这个结构的定义在 `include/linux/usb.h` 里：

```
760 struct usb_dynids {
761     spinlock_t lock;
762     struct list_head list;
763 };
```

它只有两个字段，一把锁，一个链表，都是在 `usb_register_driver()` 里面初始化的，这个 `list` 是驱动动态 `id` 链表的开头，它里面的每个节点是用另外一个结构 `struct usb_dynid` 来表示。

```
765 struct usb_dynid {
766     struct list_head node;
767     struct usb_device_id id;
768 };
```

这里面就出现了一个 `struct usb_device_id` 结构体，也就是设备的 `id`，每次添加一个动态 `id`，就会向驱动表的动态 `id` 链表里添加一个 `struct usb_dynid` 结构体。你现在应该可以想象到 `usb_match_id` 和 `usb_match_dynamic_id` 这两个函数除了查找的地方不一样，其他应该是没什么差别的。所以接下来咱们只深入探讨一下 `usb_match_id` 函数，至于 `usb_match_dynamic_id()`，它们都在 `driver.c` 中定义：

```
518 const struct usb_device_id *usb_match_id(struct usb_interface *interface,
519                                         const struct usb_device_id *id)
520 {
521     /* proc_connectinfo in devio.c may call us with id == NULL. */
522     if (id == NULL)
523         return NULL;
524
525     /* It is important to check that id->driver_info is nonzero,
526        since an entry that is all zeroes except for a nonzero
527        id->driver_info is the way to create an entry that
528        indicates that the driver want to examine every
529        device and interface. */
530     for (; id->idVendor || id->bDeviceClass || id->bInterfaceClass ||
531           id->driver_info; id++) {
532         if (usb_match_one_id(interface, id))
```

```

533         return id;
534     }
535
536     return NULL;
537 }

```

522 行，参数 `id` 指向的是驱动的设备花名册，即 `id_table`，如果它为空，那肯定就是不可能会匹配成功了。

530 行，你可能会问为什么这里不详细介绍一下 `struct usb_device_id` 结构，主要是它里面的元素的含义都相当明白。

那么这个 `for` 循环就是轮询设备花名册里的每个设备，如果符合了条件 `id->idVendor || id->bDeviceClass || id->bInterfaceClass || id->driver_info`，就调用函数 `usb_match_one_id` 做深层次的匹配。本来，在动态 `id` 出现之前这个地方是没有 `usb_match_one_id` 这个函数的，所有的匹配都在这个 `for` 循环里直接进行了，但是动态 `id` 出现之后，同时出现了前面提到的 `usb_match_dynamic_id` 函数，要在动态 `id` 链表里做同样的匹配，这就要避免代码重复，于是就将那些重复的代码提出来，组成了 `usb_match_one_id` 函数。

`for` 循环的条件里可能出现的一种情况是，`id` 的其他字段都为空，只有 `driver_info` 字段有实实在在的内容，这种情况下匹配是肯定成功的，不信的话等会儿你可以看 `usb_match_one_id()`，这种驱动对 USB 接口来说是比较随便的，不管什么接口都能和它对得上。为什么会出现这种情况？我们已经知道，接口匹配成功后，接着就会调用驱动自己的 `probe` 函数，驱动在它里面还会对接口做进一步的检查，如果真出现了这里所说的情况，意思也就是驱动将所有的检查接口，和接口培养感情的步骤都揽在自己的 `probe` 函数中了，它会在那个时候将 `driver_info` 的内容取出来，然后想怎么处理就怎么处理，本来 `id` 里边的 `driver_info` 就是给驱动保存数据用的。

还是看一看 `usb_match_one_id()` 究竟是怎么匹配的吧，定义也在 `driver.c` 里：

```

405 int usb_match_one_id(struct usb_interface *interface,
406                     const struct usb_device_id *id)
407 {
408     struct usb_host_interface *intf;
409     struct usb_device *dev;
410
411     /* proc_connectinfo in devio.c may call us with id == NULL. */
412     if (id == NULL)
413         return 0;
414
415     intf = interface->cur_altsetting;
416     dev = interface_to_usbdev(interface);
417
418     if (!usb_match_device(dev, id))
419         return 0;
420
421     /* The interface class, subclass, and protocol should never be
422      * checked for a match if the device class is Vendor Specific,
423      * unless the match record specifies the Vendor ID. */
424     if (dev->descriptor.bDeviceClass == USB_CLASS_VENDOR_SPEC &&

```

```

425             !(id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
426             (id->match_flags & (USB_DEVICE_ID_MATCH_INT_CLASS |
427             USB_DEVICE_ID_MATCH_INT_SUBCLASS |
428             USB_DEVICE_ID_MATCH_INT_PROTOCOL)))
429     return 0;
430
431     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
432         (id->bInterfaceClass != intf->desc.bInterfaceClass))
433         return 0;
434
435     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&
436         (id->bInterfaceSubClass != intf->desc.bInterfaceSubClass))
437         return 0;
438
439     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&
440         (id->bInterfaceProtocol != intf->desc.bInterfaceProtocol))
441         return 0;
442
443     return 1;
444 }

```

412 行，这个 `id` 指向的就是驱动 `id_table` 里的某一项了。

415 行，获得接口采用的设置，设置里可是有接口描述符的，要匹配接口和驱动，接口描述符里的信息是必不可少的。

416 行，从接口的 `struct usb_interface` 结构体获得 USB 设备的 `struct usb_device` 结构体，`interface_to_usbdev` 的定义在 `include/linux/usb.h` 里：

```

160 #define interface_to_usbdev(intf) \
161     container_of(intf->dev.parent, struct usb_device, dev)

```

USB 设备和它里面的接口是怎么关联起来的呢？就是上面的那个 `parent`，接口的 `parent` 早在 `usb_generic_driver` 的 `generic_probe` 函数向设备模型提交设备中的每个接口时就被初始化好了，而且指定为接口所在的 USB 设备。那么，`interface_to_usbdev` 的意思就很明显了。

418 行，这里又冒出来一个 `usb_match_device()`，接口和驱动之间的感情还真不是那么好培养的，一层一层的。不过既然存在就是有理由的，它也不会毫无根据的出现，这里虽说是在接口和接口驱动之间匹配，但是接口的 `parent` 也是必须要符合条件的，这也合情合理。所以说接口要想得到驱动，自己的 `parent` 符合驱动的条件也是很重要的，`usb_match_device()` 就是专门来匹配接口 `parent` 的。同样在 `driver.c` 中定义：

```

369 int usb_match_device(struct usb_device *dev, const struct usb_device_id *id)
370 {
371     if ((id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
372         id->idVendor != le16_to_cpu(dev->descriptor.idVendor))
373         return 0;
374
375     if ((id->match_flags & USB_DEVICE_ID_MATCH_PRODUCT) &&
376         id->idProduct != le16_to_cpu(dev->descriptor.idProduct))
377         return 0;
378

```

```

379      /* No need to test id->bcdDevice_lo != 0, since 0 is never
380         greater than any unsigned number. */
381      if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_LO) &&
382          (id->bcdDevice_lo > le16_to_cpu(dev->descriptor.bcdDevice
)))
383          return 0;
384
385      if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_HI) &&
386          (id->bcdDevice_hi < le16_to_cpu(dev->descriptor.bcdDevice
)))
387          return 0;
388
389      if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_CLASS) &&
390          (id->bDeviceClass != dev->descriptor.bDeviceClass))
391          return 0;
392
393      if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_SUBCLASS) &&
394          (id->bDeviceSubClass != dev->descriptor.bDeviceSubClass))
395          return 0;
396
397      if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_PROTOCOL) &&
398          (id->bDeviceProtocol != dev->descriptor.bDeviceProtocol))
399          return 0;
400
401      return 1;
402 }

```

这个函数采用了排比的修辞手法，美观的同时也增加了可读性。这一个个的 if 条件里都有一部分是将 id 的 match\_flags 和一个宏相与，所以弄明白 match\_flags 的意思就很关键。这里再说一下这个 match\_flags。

驱动的花名册里每个设备都对应了一个 struct usb\_device\_id 结构体，这个结构体里有很多字段，都是驱动设定好的条条框框，接口必须完全满足里面的条件才能够被驱动所接受，所以说匹配的过程就是检查接口是否满足这些条件的过程。

当然你可以每次都按照 id 的内容一个一个地比较，但是一般来说，一个驱动往往只是想设定其中的某几项，并不要求 struct usb\_device\_id 结构中的所有那些条件都要满足。

match\_flags 就是为了方便各种各样的需求而生的，驱动可以将自己的条件组合起来，match\_flags 的每一位对应一个条件，驱动在意哪个条件了，就将那一位置为 1，否则就置为 0。当然，内核中对每个驱动可能会在意的条件都定义成了宏，供驱动去组合，它们都在 include/linux/mod\_devicetable.h 中定义：

```

123 #define USB_DEVICE_ID_MATCH_VENDOR      0x0001
124 #define USB_DEVICE_ID_MATCH_PRODUCT     0x0002
125 #define USB_DEVICE_ID_MATCH_DEV_LO      0x0004
126 #define USB_DEVICE_ID_MATCH_DEV_HI      0x0008
127 #define USB_DEVICE_ID_MATCH_DEV_CLASS   0x0010
128 #define USB_DEVICE_ID_MATCH_DEV_SUBCLASS 0x0020
129 #define USB_DEVICE_ID_MATCH_DEV_PROTOCOL 0x0040
130 #define USB_DEVICE_ID_MATCH_INT_CLASS    0x0080
131 #define USB_DEVICE_ID_MATCH_INT_SUBCLASS 0x0100

```

```
132 #define USB_DEVICE_ID_MATCH_INT_PROTOCOL
```

```
0x0200
```

很容易能看出来这些数字分别表示了一个 u16 整数，也就是 `match_flags` 中的某一位。驱动比较在意哪个方面，就可以将 `match_flags` 里对应的位置为 1，在和接口匹配时自动就会去比较驱动设置的条件是否满足。那整个 `usb_match_device()` 函数就没什么说的了，就是从 `match_flags` 那里得到驱动都在意那些条件，然后将设备保存在描述符里的自身信息与 `id` 里的相应条件进行比较，有一项比较不成功就说明匹配失败，如果有一项符合了就接着看下一项，接口 `parent` 都满足条件了，就返回 1，表示匹配成功了。

还是回到 `usb_match_one_id()` 继续往下看，假设 `parent` 满足了驱动的所有条件，得以走到了 424 行。这行还要对接口的 `parent` 做点最后的确认，这行的意思就是，如果接口的 `parent`、USB 设备是属于厂商定义的 `class`，也就是不属于 `storage` 等标准的 `class`，就不再检查接口的 `class`，`subclass` 和 `protocol` 了，除非 `match_flags` 里指定了条件 `USB_DEVICE_ID_MATCH_VENDOR`。431 行之后的三个 `if` 函数也不用多说，前面是检查接口 `parent` 的，这里就是检查接口本身是不是满足驱动的条件。

当上面各个函数进行的所有检查都完全匹配时，USB 总线的 `match` 函数 `usb_device_match` 就会返回 1，表示匹配成功，之后接着就会去调用驱动的 `probe` 函数做更深入的处理，什么样的处理？这是每个驱动才知道的事情，反正到此为止，Core 的任务是已经圆满完成了，咱们的故事也就该结束了。

## 38. 结束语

这个 Core 的故事，从 `match` 开始，到 `match` 结束，在 `match` 的两端是设备和设备的驱动，是接口和接口的驱动，这个故事里遇到的人，遇到的事，早就安排在那里了，由不得我们去选择。

# 第 2 篇

## Linux 那些事儿之我是 HUB

1. 引子 .....	157	13. 再向虎山行 .....	194
2. 跟我走吧，现在就出发 .....	157	14. 树，是什么样的树 .....	198
3. 特别的爱给特别的 Root Hub .....	158	15. 没完没了的判断 .....	201
4. 一样的精灵不一样的 API .....	160	16. 一个都不能少 .....	206
5. 那些队列，那些队列操作函数 .....	164	17. 盖茨家对 Linux 代码的影响 .....	215
6. 等待，只因曾经承诺 .....	169	18. 八大重量级函数闪亮登场（一） .....	220
7. 最熟悉的陌生人——probe .....	171	19. 八大重量级函数闪亮登场（二） .....	223
8. 蝴蝶效应 .....	174	20. 八大重量级函数闪亮登场（三） .....	225
9. While You Were Sleeping（一） .....	178	21. 八大重量级函数闪亮登场（四） .....	237
10. While You Were Sleeping（二） .....	183	22. 八大重量级函数闪亮登场（五） .....	241
11. While You Were Sleeping（三） .....	185	23. 是月亮惹的祸还是 spec 的错 .....	249
12. While You Were Sleeping（四） .....	191	24. 所谓的热插拔 .....	251

---

## 1. 引子

天有不测风云，人有旦夕祸福。在 2007 年的夏天，我那可爱的电脑声卡却坏了。

朋友给我推荐了一款飞利浦的外置声卡 PSC805，老实说，声卡还能用外置的的确是我觉得新鲜，它是直接用 USB 连接，价钱也还可以。所以我去了一趟中关村买了一块外置声卡。

然而，在店家那里声卡好好的买回来之后居然连指示灯都不亮，根本没法用。不是完全不亮，一开始会亮，然后就不亮了。凭直觉，我判定这是软件的问题，而且我用的 Linux 操作系统。

从指示灯这个现象来看，我估计是电源管理部分的问题，我听说 Linux 内核 2.6.20 左右在 USB 部分开始加入了电源管理的代码，我觉得这部分代码不够成熟，问题很多也是很正常的，只是没想到我成了试验品。

我很冷静地分析一下问题，首先这块声卡是使用 USB 接口的，供电有问题那么应该是 USB 驱动部分的问题而不应该是声卡驱动的问题，声卡驱动是 `snd-usb-audio`，查看了一下日志文件，实际上问题出现在这个模块被加载之前，所以可以排除声卡驱动的问题。然后我觉得问题可能出现在 Linux 中 Hub 驱动的部分，也可能出现在主机控制器驱动的部分。这下子问题稍微麻烦了一些，我完全不清楚究竟应该分析哪个部分，于是我做了一次选择，我猜测问题会在 Hub 驱动方面，看了一下 Hub 驱动也就是三千多行代码，看了看时间，一个晚上应该是能看完的，狠一狠心，真就看了起来。

引子写到这也就该结束了，大多数人也许都会觉得我最后一定是通过看代码解决了问题，然后才会写下下面的这些文字，实际上不是的，我花了一夜看完了 Hub 驱动的代码，然而并没有发现任何异常，后来我终于知道，这个问题并不是出在 Hub 驱动的部分，它实际上与 UHCI 主机控制器的驱动代码有关，算是 UHCI 驱动的一个 Bug。但是既然我看了 Hub 驱动的代码，也不妨用文字把它记录下来，就算是为了纪念这样一个夜晚吧。

---

## 2. 跟我走吧，现在就出发

这里说的是 USB 中的 Hub。在 USB 的世界里，Hub 永远都只是绿叶，它不可能是红花，它的存在只是为了支持更多的设备连接到 USB 总线上来，谁也不会为了使用 Hub 而购买 Hub，



买 Hub 的原因是为了要使用别的设备。

也许设计代码的人和我一样，希望大家能够更多地关注 Hub，所以，关于 Hub 的代码在 Core 的目录下面。

Linux 内核代码目录中，所有设备驱动程序有关的代码都在 drivers/ 目录下面，在这个目录中的 USB 子目录包含了所有 USB 设备的驱动，而 USB 目录下面又有它自己的子目录。

注意到每一个目录下面都有一个 Kconfig 文件和一个 Makefile，这很重要。再厉害的黑客如果不看 Makefile，不看 Kconfig，也别想搞清楚这里的结构。很多年轻人喜欢研究 usb-skeleton.c，据说这个文件对他们很有启发，所以这里我也推荐一下这个文件。有时间有兴趣的话可以看一看，其实就是一个简单的 USB 设备驱动程序的框架。

执行命令 lsmod，查看它的输出，找到了 USBcore 那一行吗？Core 就是核心，基本上你要在你的电脑里用 USB 设备，那么两个模块是必需的，一个是 usbcore，这就是核心模块；另一个是主机控制器的驱动程序，比如在 usbcore 那一行看到的 ehci\_hcd 和 uhci\_hcd。

什么是 EHCI？OHCI 就是主机控制器的接口。从硬件上来说，USB 设备要想工作，除了外设本身，必须还有一个 USB 主机控制器。一般来说，一个电脑里有一个 USB 主机控制器就可以了，它就可以控制很多个设备了，比如 U 盘，USB 键盘，USB 鼠标。所有的外设都把自己的请求提交给 USB 主机控制器。然后让 USB 主机控制器统一来调度。而设备怎么连到主机控制器上？这就是我们故事的主角——Hub，“乳名”叫做集线器。

关于 Hub 的代码，在 drivers/usb/core/ 目录下面，有一个叫做 hub.c 的文件。这个文件可不简单。

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # wc -l hub.c
3122 hub.c
```

Hub 竟然有三千多行代码，真要是按行计费，写代码的那些家伙还不发财了？事实还好不是那样。

三千多行就三千多行吧，总不能见困难就退吧。跟我走吧，现在就出发。

---

## 3. 特别的爱给特别的 Root Hub

不懂 Hub 是怎么工作的就等于不知道 USB 设备驱动是怎么工作的。这句话一点没错，因为 USB 设备的初始化都是 Hub 这边发起的，通常我们写 USB 设备驱动程序都是在已经得到了一个 struct usb\_interface 指针的情况下开始 probe 工作。可是我要问你，你的 struct usb\_interface

从哪来的？老实说，要想知道从 USB 设备插入 USB 口的那一刻开始，这个世界发生了什么，你必须知道 Hub 是怎么工作的，Linux 中 Hub 驱动程序是怎么工作的。

要说 USB Hub，那得从 Root Hub 说起。什么是 Root Hub？不管你的计算机里连了多少个 USB 设备，它们最终是有根的。所有的 USB 设备最终都是连接到了一个叫做 Root Hub 的设备上，或者说所有的根源都是从这里开始的。

Root Hub 上可以连接别的设备，可以连接 U 盘，可以连接 USB 鼠标，同样也可以连接另一个 Hub。所谓 Hub，就是用来级连。但是普通的 Hub，它一头接上级 Hub，另一头可以有多个口，多个口就可以级连多个设备，也可以只有一个口。而 Root Hub 呢？它比较特殊，它当然也是一种 USB 设备，但是它属于一人之下万人之上的角色，它只属于主机控制器，换言之，通常做芯片的人会把主机控制器和 Root Hub 集成在一起。特别是 PC 主机上，通常你就只能看到接口，看不到 Root Hub，因为它在主机控制器里，正如图 2.3.1 所示。

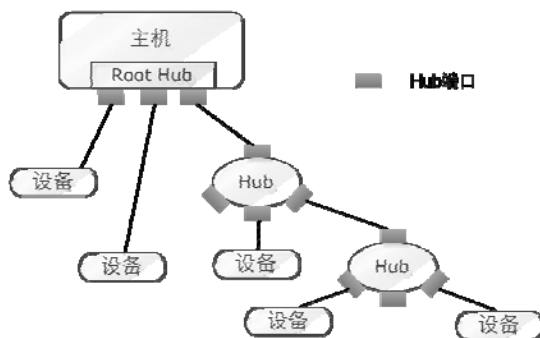


图 2.3.1 USB 拓扑结构

当然，我们应该更加准确地评价主机和 Root Hub 的关系。

别扯远了，继续回来，既然 Root Hub 享有如此特殊的地位，那么很显然，整个 USB 子系统得特别对待它，一开始就会要初始化 Hub。所以我们从 USB 子系统的初始化代码开始看起，也就是 `usb_init` 函数，来自 `drivers/usb/core/usb.c`：

```

863 static int __init usb_init(void)
864 {
    □□
890     if (retval)
891         goto fs_init_failed;
892     retval = usb_hub_init();
    □□
916 }

```

在众多名叫 `*init*` 的函数之中，是有一个属于 Hub 的，它在这里初始化，换言之，随着 USB Core 的初始化，Hub 也就开始了它的初始化之路，`usb_hub_init()` 函数得到调用，这个函数

来自 drivers/usb/core/hub.c:

```
2854 int usb_hub_init(void)
2855 {
2856     if (usb_register(&hub_driver) < 0) {
2857         printk(KERN_ERR "%s: can't register hub driver\n",
2858             usbcore_name);
2859         return -1;
2860     }
2861
2862     khubd_task = kthread_run(hub_thread, NULL, "khubd");
2863     if (!IS_ERR(khubd_task))
2864         return 0;
2865
2866     /* Fall through if kernel_thread failed */
2867     usb_deregister(&hub_driver);
2868     printk(KERN_ERR "%s: can't start khubd\n", usbcore_name);
2869
2870     return -1;
2871 }
```

一路走来的兄弟们不会对这样一段代码陌生，是不是有一种似曾相识的感觉？

---

## 4. 一样的精灵不一样的 API

usb\_register()这个函数是用来向 USB 核心层，即 USB Core，注册一个 USB 设备驱动的，而这里我们注册的是 Hub 的驱动程序所对应的 struct usb\_driver 结构体变量。定义于 drivers/usb/core/hub.c 中：

```
2841 static struct usb_driver hub_driver = {
2842     .name = "hub",
2843     .probe = hub_probe,
2844     .disconnect = hub_disconnect,
2845     .suspend = hub_suspend,
2846     .resume = hub_resume,
2847     .pre_reset = hub_pre_reset,
2848     .post_reset = hub_post_reset,
2849     .ioctl = hub_ioctl,
2850     .id_table = hub_id_table,
2851     .supports_autosuspend = 1,
2852 };
```

这里面最重要的一个函数就是 hub\_probe，很多事情都在这期间发生了。每个 USB 设备（或者说所有设备）的驱动都会有一个 probe 函数，比如 U 盘的 probe 函数就是 storage\_probe()，不过 storage\_probe()被调用需要有两个前提，第一个前提是 usb-storage 被加载了，第二个前提是 U 盘等设备插入了被检测到了。

而 Hub，说它特别，我可绝不是“忽悠”你。Hub 本身就有两种，一种是普通的 Hub，一

种是 Root Hub。对于普通 Hub，它完全可能也是和 U 盘一样，在某个时刻被插入，然后在这种情况下 `hub_probe` 被调用，但是对于 Root Hub 就不需要这么多废话了，Root Hub 肯定是有的，只要你有 USB 主机控制器，就一定要有 Root Hub，所以 `hub_probe()` 基本上是很自然地被调用了，不用说非得等待某个插入事件的发生，没这个必要。当然没有 USB 主机控制器就没有 USB 设备能工作。那么 USB Core 这整个模块你就没有必要分析了。所以，只要你有 USB 主机控制器，那么在 USB 主机控制器的驱动程序初始化的过程中，它就会调用 `hub_probe()` 来探测 Root Hub，不管你的主机控制器是 OHCI、UHCI 还是 EHCI 的接口。

如果 `register` 一切顺利的话，那么返回值为 0。如果返回值为负数，就说明出错了。现在假设这一步没有出错。

`usb_hub_init()` 的 2862 行，这行代码其实是很有技术含量的，不过对于写驱动的人来说，其作用就和当年的 `kernel_thread()` 相当。不过 `kernel_thread()` 返回值是一个 `int` 型的，而 `kthread_run()` 返回的却是 `struct task_struct` 结构体指针。这里等号左边的 `khubd_task` 是我们自己定义的一个 `struct task_struct` 指针：

```
88 static struct task_struct *khubd_task;
```

`struct task_struct` 不用多说，记录进程的数据结构。每一个进程都用一个 `struct task_struct` 结构体变量来表示。所以这里所做的就是记录下创建好的内核进程，以便日后要卸载模块时可以用另一个函数来结束这个内核进程（你也可以叫内核线程），到时我们会调用 `kthread_stop(khubd_task)` 函数来结束这个内核线程，这个函数的调用我们将会在 `usb_hub_cleanup()` 函数中看到。而 `usb_hub_cleanup()` 正是 Hub 里面和 `usb_hub_init()` 相对应的函数。

2863 行，判断一下 `khubd_task`，`IS_ERR` 是一个宏，用来判断指针的。当你创建了一个进程，你当然想知道这个进程创建成功了没有。

以前我们注意到每次申请内存时都会做一次判断，你说创建进程是不是也要申请内存？不申请内存谁来记录 `struct task_struct`？很显然，要进行判断。以前我们判断的是指针是否为空。以后接触代码多了你会发现，其实 Linux 内核中有很多种内存申请的方式，而这些方式所返回的内存地址也是不一样的，所以并不是每一次我们都只要判断指针是否为空就可以了。事实上，每一次调用 `kthread_run()` 之后，我们都会用一个 `IS_ERR()` 来判断指针是否有效。`IS_ERR()` 为 1 就表示指针有错，或者准确一点说叫做指针无效。

什么叫指针无效？后面会专门会解释，让我们继续往下看，只需要记得，如果你不希望发生缺页异常这样的错误的话，每次调用完 `kthread_run()` 之后要用 `IS_ERR()` 来检测一下返回的指针。如果 `IS_ERR()` 返回值是 0，那么说明没有问题，于是返回值为 0，也就是说 `usb_hub_init()` 就这么结束了。反之，就会执行 `usb_deregister()`，因为内核线程没有成功创建，hub 就没法驱动起来了。

最后函数在 2870 行，返回值为 -1。回到 `usb_init()` 函数中我们会知道，接下来 `usb_hub_cleanup()` 就会被调用。`usb_hub_cleanup()` 同样定义于 `drivers/usb/core/hub.c` 中：

```
2873 void usb_hub_cleanup(void)
2874 {
2875     kthread_stop(khubd_task);
2876
2877     /*
2878      * Hub resources are freed for us by usb_deregister. It calls
2879      * usb_driver_purge on every device which in turn calls that
2880      * devices disconnect function if it is using this driver.
2881      * The hub_disconnect function takes care of releasing the
2882      * individual hub resources. -greg
2883      */
2884     usb_deregister(&hub_driver);
2885 } /* usb_hub_cleanup() */
```

这个函数我想没有任何必要解释了吧。`kthread_stop()` 和刚才的 `kthread_run()` 对应，`usb_deregister()` 和 `usb_register()` 对应。

总之，如果创建子进程出了问题，那么一切都免谈。

反之，如果成功了，那么 `kthread_run()` 的三个参数就是我们要关注的了，第一个是 `hub_thread()`，子进程将从这里开始执行。第二个是 `hub_thread()`，传递的是 `NULL`，第三个参数就是精灵进程的名字 `ps -el` 看一下，如下所示：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ps -el | grep khubd
1 S      0 1963    27 0 70 -5 -      0 hub_th ?      00:00:00 khubd
```

你就会发现有这么一个精灵进程运行着。所以，下一步，让我们进入 `hub_thread()` 来查看这个子进程吧。

以下是关于 `IS_ERR` 的介绍文字。如果你对内存管理没有任何兴趣，就不用往下看了。要想明白 `IS_ERR()`，首先你得知道有一种空间叫做内核空间，不清楚也不要紧。结合 `IS_ERR()` 的代码来看，来自 `include/linux/err.h`：

```
16 #define MAX_ERRNO      4095
17
18 #ifndef __ASSEMBLY__
19
20 #define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
21
22 static inline void *ERR_PTR(long error)
23 {
24     return (void *) error;
25 }
26
27 static inline long PTR_ERR(const void *ptr)
28 {
29     return (long) ptr;
30 }
31
```

```

32 static inline long IS_ERR(const void *ptr)
33 {
34     return IS_ERR_VALUE((unsigned long)ptr);
35 }
36
37 #endif

```

关于内核空间，我只想说，所有的驱动程序都是运行在内核空间，内核空间虽然很大，但是总是有限的。而在这有限的空间中，其最后一个 page 是专门保留的，也就是说一般人不可能用到内核空间最后一个 page 的指针。

换句话说，你在写设备驱动程序的过程中，涉及的任何一个指针，必然有三种情况，一种是有有效指针，一种是 NULL（空指针），还有一种是错误指针，或者说无效指针。而所谓的错误指针就是指其已经到达了最后一个 page。比如对于 32bit 的系统来说，内核空间最高地址 0xffffffff，那么最后一个 page 就是指的 0xffff000~0xffffffff(假设 4KB 一个 page)。这段地址是被保留的，一般人不得越雷池半步，如果你发现你的一个指针指向这个范围中的某个地址，那么恭喜你，你的代码肯定出错了。

那么你是不是很好奇，好端端的内核空间干嘛要留出最后一个 page？这不是明明自己有 1000 块钱，非得对自己说只能用 900 块。实在不好意思，你说错了，这里不仅不是浪费一个 page，反而是充分利用资源，把一个东西当两个东西来用。

看见 16 行那个“MAX\_ERRNO”了吗？一个宏，定义为 4095，MAX\_ERRNO 就是最大错误号，Linux 内核中，出错有多种可能，因为有许多多种错误。关于 Linux 内核中的错误，我们看一下 include/asm-generic/errno-base.h 文件：

```

#define EPERM          1      /* Operation not permitted */
#define ENOENT          2      /* No such file or directory */
#define ESRCH          3      /* No such process */
#define EINTR          4      /* Interrupted system call */
#define EIO            5      /* I/O error */
#define ENXIO          6      /* No such device or address */
#define E2BIG          7      /* Argument list too long */
#define ENOEXEC        8      /* Exec format error */
#define EBADF          9      /* Bad file number */
#define ECHILD         10     /* No child processes */
#define EAGAIN         11     /* Try again */
#define ENOMEM         12     /* Out of memory */
#define EACCES         13     /* Permission denied */
#define EFAULT         14     /* Bad address */
#define ENOTBLK        15     /* Block device required */
#define EBUSY          16     /* Device or resource busy */
#define EEXIST         17     /* File exists */
#define EXDEV          18     /* Cross-device link */
#define ENODEV         19     /* No such device */
#define ENOTDIR        20     /* Not a directory */
#define EISDIR         21     /* Is a directory */
#define EINVAL         22     /* Invalid argument */
#define ENFILE         23     /* File table overflow */
#define EMFILE         24     /* Too many open files */

```

```

#define ENOTTY      25      /* Not a typewriter */
#define ETXTBSY     26      /* Text file busy */
#define EFBIG       27      /* File too large */
#define ENOSPC      28      /* No space left on device */
#define ESPIPE      29      /* Illegal seek */
#define EROFS       30      /* Read-only file system */
#define EMLINK      31      /* Too many links */
#define EPIPE       32      /* Broken pipe */
#define EDOM        33      /* Math argument out of domain of func */
#define ERANGE      34      /* Math result not representable */

```

最常见的几个是-EBUSY、-EINVAL、-ENODEV、-EPIPE、-EAGAIN、-ENOMEM，我相信只要你使用过 Linux 就有可能见过这几个错误，因为它们确实经常出现。

这些是每个体系结构中都有，另外各个体系结构也都定义了自己的一些错误代码。这些东西当然也都是宏，实际上对应的是一些数字，这些数字就叫做错误号。而对于 Linux 内核来说，不管任何体系结构，错误号最多不会超过 4095，而 4095 又正好是比 4KB 小 1，即 4096-1。而我们知道一个 page 可能是 4KB，也可能是更多，比如 8KB，但至少它也是 4KB，所以留出一个 page 出来就可以让我们把内核空间的指针来记录错误了。

什么意思呢？比如我们这里的 IS\_ERR()，它就是判断 kthread\_run() 返回的指针是否有错，如果指针并不是指向最后一个 page，那么没有问题，申请成功了，如果指针指向了最后一个 page，那么说明实际上这不是一个有效的指针，这个指针里保存的实际上是一种错误代码。而通常很常用的方法就是先用 IS\_ERR() 来判断是否是错误，然后如果是，那么就调用 PTR\_ERR() 来返回这个错误代码。只不过这里没有调用 PTR\_ERR() 而已，因为起决定作用的还是 IS\_ERR()，而 PTR\_ERR() 只是返回错误代码，也就是提供一个信息给调用者，如果你只需要知道是否出错，而不在乎因为什么而出错，那你当然不用调用 PTR\_ERR() 了。当然，这里如果出错了的话，最终 usb\_deregister() 会被调用，并且 usb\_hub\_init() 会返回-1。

## 5. 那些队列，那些队列操作函数

这一节我们讲队列。

随着子进程进入了我们的视野，我们来看其入口函数 hub\_thread()，这是一个令你大跌眼镜的函数。

```

2817 static int hub_thread(void *__unused)
2818 {
2819     do {
2820         hub_events();
2821         wait_event_interruptible(khubd_wait,
2822                                 !list_empty(&hub_event_list) ||
2823                                 kthread_should_stop());

```

```

2824         try_to_freeze();
2825 } while (!kthread_should_stop() || !list_empty(&hub_event_list));
2826
2827     pr_debug("%s: khubd exiting\n", usbcore_name);
2828     return 0;
2829 }

```

这就是 Hub 驱动中最精华的代码。这几乎是一个死循环，但是关于 Hub 的所有故事都发生在这里，没错，就在这短短几行代码中。

而这其中，最核心的函数自然是 `hub_events()`。我们先不看 `hub_events()`，先把外面这几个函数看明白了。`kthread_should_stop()` 的意思很明显，就是字面意思——是不是该停掉。如果是，那么这里循环就结束了，`hub_thread()` 返回 0，而要让 `kthread_should_stop()` 为真，就是当我们调用 `kthread_stop()` 时。这种情况，这个进程就该结束了。

再看 `hub_event_list`，同样来自 `drivers/usb/core/hub.c`:

```

83 static LIST_HEAD(hub_event_list); /* List of hubs needing servicing */

```

我们来看一下 `LIST_HEAD` 吧，当你越接近那些核心的代码，你就会发现关于链表的定义就会越多。其实在 `usb-storage` 里面，我们也提到过一些链表，但却并没有自己用 `LIST_HEAD` 来定义过链表，因为我们用不着。可是 Hub 这边就有用了，当然主机控制器的驱动程序里也会有。

使用链表的目的很明确，因为有很多事情要做，于是就把它放进链表里，一件事一件事地处理。还记得我们当初在 `usb-storage` 里面提交 `urb` 请求了吗？你的 U 盘不停地提交 `urb` 请求，USB 键盘也提交，USB 鼠标也提交，那 USB 主机控制器怎么能应付得过来呢？很简单，建立一个队列，然后你每次提交就是往一个队列里边插入，然后 USB 主机控制器再统一去调度，一个一个来执行。

那么这里 Hub 它有什么事件？比如探测到一个设备连进来了，于是就会执行一些代码去初始化设备，所以就建一个队列。关于 Linux 内核中的链表，可以专门写一篇文章了，我们简单介绍一下，来看 `include/linux/list.h`:

```

21 struct list_head {
22     struct list_head *next, *prev;
23 };
24
25 #define LIST_HEAD_INIT(name) { &(amp;name), &(name) }
26
27 #define LIST_HEAD(name) \
28     struct list_head name = LIST_HEAD_INIT(name)

```

可以看出，我们无非就是定义了一个 `struct list_head` 的结构体 `hub_event_list`，而且其两个指针 `next` 和 `prev` 分别是指向自己。换言之，我们建立了一个链表，而且是双向链表，并且做了初始化，初始化成一个空队列。而对于这个队列的操作，内核提供了很多函数可以使用，不



过在 Hub 中，我们将会用到这么几个函数。

```
298 static inline int list_empty(const struct list_head *head)
299 {
300     return head->next == head;
301 }
```

不言自明，判断队列是否为空。我们说了，初始化时这个 `hub_event_list` 队列是空的。

有了队列，自然就要操作队列，要往队列里加东西、减东西。就像我们每个人每天都在不停地走进，又走出。来看第二个函数 `list_add_tail()`，这就是往队列里加东西：

```
84 static inline void list_add_tail(struct list_head *new, struct list_head
*head)
85 {
86     __list_add(new, head->prev, head);
87 }
```

继续跟着 `__list_add()` 看就会发现其实就是往队列的末尾加一个元素：

```
43 static inline void __list_add(struct list_head *new,
44                               struct list_head *prev,
45                               struct list_head *next)
46 {
47     next->prev = new;
48     new->next = next;
49     new->prev = prev;
50     prev->next = new;
51 }
```

再来看下一个 `list_del_init()`，队列里的元素不能只加不减，没用了的元素就该删除掉，把空间腾出来给别人：

```
254 static inline void list_del_init(struct list_head *entry)
255 {
256     __list_del(entry->prev, entry->next);
257     INIT_LIST_HEAD(entry);
258 }
```

从队列里删除一个元素，并且将该元素做初始化，首先看 `__list_del()`：

```
155 static inline void __list_del(struct list_head * prev, struct list_head *
next)
156 {
157     next->prev = prev;
158     prev->next = next;
159 }
```

`INIT_LIST_HEAD()` 其实就是初始化为最初的状态，即一个空链表。如下：

```
30 static inline void INIT_LIST_HEAD(struct list_head *list)
31 {
32     list->next = list;
33     list->prev = list;
34 }
```

当然了，还有一个超级经典的 `list_entry()`，和以上所有 `list` 方面的宏一样，也是来自 `include/linux/list.h`：

```
425 #define list_entry(ptr, type, member) \
426     container_of(ptr, type, member)
```

我相信，`list_entry()`这个宏在 Linux 内核代码中的地位都是耳熟能详的，如果你说你不知道 `list_entry()`，那你千万别跟人说你懂 Linux 内核。

`list_entry` 这个宏应该说还是有一定技术含量的。

关于 `list_entry()`，让我们结合实例来看，我们在 `hub` 的故事里会接触到的一个重要的数据结构就是 `struct usb_hub`，来自 `drivers/usb/core/hub.c`：

```
34 struct usb_hub {
35     struct device          *intfdev;      /* the "interface" device */
36     struct usb_device      *hdev;
37     struct urb             *urb;         /* for interrupt polling pipe */
38
39     /* buffer for urb ... with extra space in case of babble */
40     char                    (*buffer)[8];
41     dma_addr_t             buffer_dma;    /* DMA address for buffer */
42     union {
43         struct usb_hub_status  hub;
44         struct usb_port_status port;
45     } *status;      /* buffer for status reports */
46     struct mutex            status_mutex; /* for the status buffer */
47
48     int                    error;        /* last reported error */
49     int                    nerrors;     /* track consecutive errors */
50
51     struct list_head       event_list;   /* hubs w/data or errs ready */
52     unsigned long          event_bits[1]; /* status change bitmask */
53     unsigned long          change_bits[1]; /* ports with logical connect
54                                           status change */
55     unsigned long          busy_bits[1]; /* ports being reset or
56                                           resumed */
57 #if USB_MAXCHILDREN > 31 /* 8*sizeof(unsigned long) - 1 */
58 #error event_bits[] is too short!
59 #endif
60
61     struct usb_hub_descriptor *descriptor; /* class descriptor */
62     struct usb_tt            tt;           /* Transaction Translator */
63
64     unsigned                mA_per_port; /* current for each child */
65
66     unsigned                limited_power:1;
67     unsigned                quiescing:1;
68     unsigned                activating:1;
69
70     unsigned                has_indicators:1;
71     u8                      indicator[USB_MAXCHILDREN];
72     struct delayed_work     leds;
73 };
```

看到了吗，51 行，`struct list_head event_list`，这个结构体变量将为我们组建一个队列，或者说组建一个链表。我们知道，一个 `struct usb_hub` 代表一个 Hub，每一个 `struct usb_hub` 有一个 `event_list`，即每一个 Hub 都有它自己的一个事件列表。要知道 Hub 可以有一个或者有多个，而 Hub 驱动只需要一个，或者说 `khubd` 这个精灵进程永远都只有一个。而我们的做法是，不管实际上有多少个 Hub，我们最终都会将其 `event_list` 挂入到全局链表 `hub_event_list` 中来统一处理（`hub_event_list` 对于整个 USB 系统来说是全局的，但对于整个内核来说当然是局部的，毕竟它前面有一个 `static`）。

因为最终处理所有 Hub 的事务的都是我们这一个精灵进程 `khubd`，它只需要判断 `hub_event_list` 是否为空，不为空就去处理。或者说就去触发 `hub_events()` 函数，但当我们真的要处理 Hub 的事件时，我们当然要知道具体是哪个 Hub 触发了这起事件。而 `list_entry` 的作用就是，从 `struct list_head event_list` 得到它所对应的 `struct usb_hub` 结构体变量。比如以下的四行代码：

```
struct list_head *tmp;
struct usb_hub *hub;
tmp=hub_event_list.next;
hub=list_entry(tmp,struct usb_hub,event_list);
```

从全局链表 `hub_event_list` 中取出一个来，叫做 `tmp`，然后通过 `tmp` 获得它所对应的 `struct usb_hub`。

最后，总结一下，中学我们学习写议论文时，老师教过有这样几种结构：总分总式结构，对照式结构，层进式结构，并列式结构。而总分总式结构就是先提出中心论点，然后围绕中心，以不同角度提出分论点，展开论述，最后进行总结。而总分总具体来说又有总分，分总，总分总三种形式。

以前我以为 `Linux` 只是技术比我强，现在我算是看明白了，语文学得也比我好。看出来了吗？这里采用的就是我们议论文中的总分总结构，先设置一个链表 `hub_event_list`，设置一个总的函数 `hub_events()`，这是“总”；然后每一个 Hub 都有一个 `event_list`，每当有一个 hub 的 `event_list` 出现了变化，就把它的 `event_list` 插入到 `hub_event_list` 中来，这是“分”；然后触发总函数 `hub_events()`，这又是“总”；然后在 `hub_events()` 里又根据 `event_list` 来确定是哪个 `struct usb_hub`，或者说是哪个 Hub 有问题，又针对该 Hub 进行具体处理，这又是“分”。这就是 `Linux` 中 Hub 驱动的中心思想。

最后，提醒一下各位读者，`struct usb_hub` 在这里介绍了，稍后讲到这个结构体时就不会再介绍了。

## 6. 等待，只因曾经承诺

hub\_thread()中还有一个函数没有讲，它就是 try\_to\_freeze()，这是与电源管理相关的函数。对大多数人来说，关于这个函数，了解就可以了。

随着 Linux 开始支持 suspended 之后，有人提倡，每一个内核进程都应该在适当的时候，调用 try\_to\_freeze()。什么意思呢？有这样一个 flag，PF\_NOFREEZE，如果你这个进程或者内核线程不想进入 suspended 状态，那么你就可以设置这个 flag，正如我们在 usb-storage 中 usb\_stor\_control\_thread()中做的那样。而对于大多数内核线程来说，目前主流的看法是希望你能在某个地方调用 try\_to\_freeze()，这个函数的作用是检测一个 flag 有没有设置，哪个 flag 呢，TIF\_FREEZE，每个体系结构定义了自己与这有关的 flags，比如 i386 的，include/asm-i386/thread\_info.h 中：

```
126 #define TIF_SYSCALL_TRACE      0      /* syscall trace active */
127 #define TIF_NOTIFY_RESUME      1 /* resumption notification requested */
128 #define TIF_SIGPENDING         2      /* signal pending */
129 #define TIF_NEED_RESCHED       3      /* rescheduling necessary */
130 #define TIF_SINGLESTEP 4 /* restore singlestep on return to user mode*/
131 #define TIF_IRET                5      /* return with iret */
132 #define TIF_SYSCALL_EMU         6      /* syscall emulation active */
133 #define TIF_SYSCALL_AUDIT       7      /* syscall auditing active */
134 #define TIF_SECCOMP             8      /* secure computing */
135 #define TIF_RESTORE_SIGMASK     9 /* restore signal mask in do_signal() */
136 #define TIF_MEMDIE              16
137 #define TIF_DEBUG               17     /* uses debug registers */
138 #define TIF_IO_BITMAP           18     /* uses I/O bitmap */
139 #define TIF_FREEZE              19     /* is freezing for suspend */
```

一句话，如果你不想支持电源管理，那么你编译内核时把 CONFIG\_PM 给关了。不过，有一个问题，USB 设备实际上是有节电这个特性的，也就是说 USB 的各种规范中就有一个 suspend 和一个 resume，也就是挂起和恢复，换言之，硬件本身有这样的特性，要是软件不支持的话写出来的代码你敢给客户用吗？不过一个利好消息是，除了这里这个 try\_to\_freeze()比较难一点外，剩下的在 USB 中出现的电源管理的代码实际上相对来说不是很难理解，毕竟那些东西和硬件规范是对应的，都有章可循，硬件怎么规定就怎么做，所以，不用太担心。

摆平了外面的这行代码，于是现在我们安心来看 hub\_events()了。hub\_events()还是来自 drivers/usb/core/hub.c，我们一段一段地来看。

```
2595 static void hub_events(void)
2596 {
2597     struct list_head *tmp;
2598     struct usb_device *hdev;
2599     struct usb_interface *intf;
2600     struct usb_hub *hub;
2601     struct device *hub_dev;
2602     u16 hubstatus;
2603     u16 hubchange;
2604     u16 portstatus;
```

```

2605     u16 portchange;
2606     int i, ret;
2607     int connect_change;
2608
2609     /*
2610      * We restart the list every time to avoid a deadlock with
2611      * deleting hubs downstream from this one. This should be
2612      * safe since we delete the hub from the event list.
2613      * Not the most efficient, but avoids deadlocks.
2614      */
2615     while (1) {
2616
2617         /* Grab the first entry at the beginning of the list */
2618         spin_lock_irq(&hub_event_lock);
2619         if (list_empty(&hub_event_list)) {
2620             spin_unlock_irq(&hub_event_lock);
2621             break;
2622         }
2623
2624         tmp = hub_event_list.next;
2625         list_del_init(tmp);
2626
2627         hub = list_entry(tmp, struct usb_hub, event_list);
2628         hdev = hub->hdev;
2629         intf = to_usb_interface(hub->intfdev);
2630         hub_dev = &intf->dev;
2631
2632         dev_dbg(hub_dev, "state %d ports %d chg %04x evt %04x\n",
2633                 hdev->state, hub->descriptor
2634                     ? hub->descriptor->bNbrPorts
2635                     : 0,
2636                 /* NOTE: expects max 15 ports... */
2637                 (u16) hub->change_bits[0],
2638                 (u16) hub->event_bits[0]);
2639
2640         usb_get_intf(intf);
2641         spin_unlock_irq(&hub_event_lock);

```

2615 行，一个 while(1) 循环；2619 行，判断 hub\_event\_list 是否为空，是不是觉得很有趣？第一次调用这个函数时，hub\_event\_list 就是初值，我们说过初值为空，所以这里就是空，即 list\_empty() 返回 1，然后 break 语句跳出 while 循环。你知道 while 循环的结尾在哪里吗？就是这个 hub\_events() 函数的结尾，也就是说在这里几百行的代码就结束了，我们直接退出这个函数，返回到 hub\_thread() 中，调用 wait\_event\_interruptible() 进入睡眠，然后等待有事件发生。

对于 Hub 来说，当你插入一个设备到 Hub 口里，就会触发一事件。而第一事件的发生其实是 Hub 驱动程序本身的初始化，即我们说过，由于 Root Hub 的存在，所以 hub\_probe 必然会被调用，确切地说，就是在主机控制器的驱动程序中，一定会调用 hub\_probe 的。如果你问我到底什么时候会调用，那么我无可奉告，因为这是在主机控制器的驱动程序中，不管你的主机控制器是属于 OHCI 的、UHCI 的，还是 EHCI 的，最终在它们的初始化代码中都会调用一个叫做 hcd\_register\_root() 的函数，进而转到 usb\_register\_root\_hub()，几经周转，最终 hub\_probe 就会被调用。所以你根本不用担心这个函数什么时刻会被调用，反正总会有这个时刻。

所以，我们就转到 `hub_probe` 吧，这里 `hub_events()` 只是虚晃一枪，不过你别忘了，等到 `hub_event_list` 里面有东西了之后，我们还会回来的。要知道 `hub_events()` 这个函数才是真正的 Hub 驱动的核心函数，所有的故事都是在这里发生的。所以，就像你给了某人一个承诺，承诺你还会回来。有了承诺，等待也被赋予了意义。

最后需要记住的是 `wait_event_interruptible()` 的第一个参数是 `&khubd_wait`，关于这个函数我们在 `usb-storage` 里面已经看过多次了，其中 `khubd_wait` 定义于 `drivers/usb/core/hub.c`：

```
85 /* Wakes up khubd */
86 static DECLARE_WAIT_QUEUE_HEAD(khubd_wait);
```

这无非就是一个等待队列头，所以我们很清楚，将来要唤醒这个睡眠进程的一定是类似这样的一行代码：`wake_up(&khubd_wait)`。没错，整个内核代码中只有一个地方会调用这个代码，那就是 `kick_khubd()`，不过调用 `kick_khubd()` 的地方可不少。

## 7. 最熟悉的陌生人——probe

话说因为 Hub 驱动无所事事，所以 `hub_thread()` 进入了睡眠，直到某一天，`hub_probe` 被调用。所以我们来看一下 `hub_probe()`，这个函数来自 `drivers/usb/hub.c`，其作用就如同当初我们在 `usb-storage` 中遇到的那个 `storage_probe()` 函数一样。

```
887 static int hub_probe(struct usb_interface *intf, const struct usb_device_id
*id)
888 {
889     struct usb_host_interface *desc;
890     struct usb_endpoint_descriptor *endpoint;
891     struct usb_device *hdev;
892     struct usb_hub *hub;
893
894     desc = intf->cur_altsetting;
895     hdev = interface_to_usbdev(intf);
896
897 #ifdef CONFIG_USB_OTG_BLACKLIST_HUB
898     if (hdev->parent) {
899         dev_warn(&intf->dev, "ignoring external hub\n");
900         return -ENODEV;
901     }
902 #endif
903
904     /* Some hubs have a subclass of 1, which AFAICT according to the */
905     /* specs is not defined, but it works */
906     if ((desc->desc.bInterfaceSubClass != 0) &&
907         (desc->desc.bInterfaceSubClass != 1)) {
908     descriptor_error:
909         dev_err (&intf->dev, "bad descriptor, ignoring hub\n");
910         return -EIO;
911     }
```

```

912
913     /* Multiple endpoints? What kind of mutant ninja-hub is this? */
914     if (desc->desc.bNumEndpoints != 1)
915         goto descriptor_error;
916
917     endpoint = &desc->endpoint[0].desc;
918
919     /* If it's not an interrupt in endpoint, we'd better punt! */
920     if (!usb_endpoint_is_int_in(endpoint))
921         goto descriptor_error;
922
923     /* We found a hub */
924     dev_info (&intf->dev, "USB hub found\n");
925
926     hub = kzalloc(sizeof(*hub), GFP_KERNEL);
927     if (!hub) {
928         dev_dbg (&intf->dev, "couldn't kmalloc hub struct\n");
929         return -ENOMEM;
930     }
931
932     INIT_LIST_HEAD(&hub->event_list);
933     hub->intfdev = &intf->dev;
934     hub->hdev = hdev;
935     INIT_DELAYED_WORK(&hub->leds, led_work);
936
937     usb_set_intfdata (intf, hub);
938     intf->needs_remote_wakeup = 1;
939
940     if (hdev->speed == USB_SPEED_HIGH)
941         highspeed_hubs++;
942
943     if (hub_configure(hub, endpoint) >= 0)
944         return 0;
945
946     hub_disconnect (intf);
947     return -ENODEV;
948 }

```

幸运的是这个函数还不是很长。894 行，`desc`，是这个函数中定义的一个 `struct usb_host_interface` 结构体指针，其实这就相当于 `struct usb_interface` 结构中的那个 `altsetting`，只是换了一个名字。

同样 895 行这个赋值我们也是很眼熟，`interface_to_usbdev()` 这个宏就是为了从一个 `struct usb_interface` 的结构体指针得到那个与它相关的 `struct usb_device` 结构体指针。这里等号右边的 `intf` 自不必说，而左边的 `hdev` 正是我们这里为了 Hub 而定义的一个 `struct usb_device` 结构体指针。

897 行到 902 行，这是为 OTG 而准备的，为了简化问题，在这里我做一个假设，即假设我们不支持 OTG。在内核编译选项中有一个叫做 `CONFIG_USB_OTG` 的选项，OTG 就是“On The Go”（正在进行中）的意思，随着 USB 传输协议的诞生，以及它的迅速走红，人们不再满足于以前那种一个设备要么就是主设备，要么就是从设备的现状，也就是说要么是 Host（或者叫主设备）；要么是外设（也叫 Slave，或者叫从设备）。在那个年代里，只有当一台 Host 与一台 Slave

连接时才能实现数据的传输，而后来开发人员们又公布了 USB OTG 规范，于是出现了 OTG 设备，即既可以充当 Host，亦能充当 Slave 的设备。也就是说如果你有一台数码相机和一台打印机，它们各有一个 USB 接口，把这两个口连接起来，就可以把你的照片打印出来了。所以我只能假设我们不打开支持 OTG 的编译开关，而这里我们看到的 CONFIG\_USB\_OTG\_BLACKLIST\_HUB，其实就是 CONFIG\_OTG 下面的子选项，不选后者根本就见不到前者。

904 行到 911 行，这没什么可说的了，每一个 USB 设备它属于哪个类，以及哪个子类这都是定好的，比如 Hub 的子类就是 0，即 desc->desc 这个 interface 描述符里边的 bInterfaceSubClass 就应该是 0。所以这里是判断如果 bInterfaceSubClass 不为 0 那就出错了，那就不往下走了，返回值是-EIO。

914 行和 915 行，其实干的事情是差不多的，针对接口描述符再做一次判断，这次是判断这个 Hub 有几个端点。spec 规定了 Hub 只有一个端点（除去端点 0）也就是中断端点，因为 Hub 的传输是中断传输。当然还有控制传输，但是因为控制传输是每一个设备都必须支持的，即每一个 USB 设备都会有一个控制端点，所以在 desc->desc.bNumEndpoints 中是不包含那个大家都有的控制端点的。因此如果这个值不为 1，那么就说明又出错了，仍然只能是返回。

917 行，得到这个唯一的端点所对应的端点描述符，920 行和 921 行就是判断这个端点是不是中断端点，如果不是，那还是一样，返回报错吧。

如果以上几种常见的错误都没有出现，这个时候我们才开始正式地去做一些事情，让我们继续。

924 行，打印调试信息。

926 行，申请 Hub 的数据结构 struct usb\_hub。不过 926 行有一个很新的函数，kzalloc()。其实这个函数就是原来的两个函数的整合，即原来我们每次申请内存时都会这么做，先是用 kmalloc()申请空间，然后用 memset()来初始化，而现在一步到位，直接调用 kzalloc()函数，效果等同于原来那两个函数，所有申请的元素都被初始化为 0。

其实对于写驱动的人来说，知道现在应该用 kzalloc()函数代替原来的 kmalloc()和 memset()函数就可以了，这是内核中内存管理部分做出的改变，确切地说是改进。负责内存管理那部分程序的目标无非就是让内核跑起来更快一些，而从 kmalloc/memset 到 kzalloc 的改变确实也是为了实现这方面的优化。所以自从 2005 年底内核中引入 kzalloc 之后，整个内核代码的许多模块里面都先后把原来的 kmalloc/memset 统统换成了 kzalloc()。咱们这里就是其中一处。927 行到 930 行不用说了，如果没申请成功那就返回 ENOMEM。

932 行，还记得之前说的总分的结构，一个总的事件队列，hub\_event\_list，然后各个 Hub 都有一个分的事件队列，就是这里的 hub->event\_list，前面已经初始化了全局的 hub\_event\_list，



而这里咱们针对单个 Hub 就得为其初始化一个 `event_list`。

933 行和 934 行, `struct usb_hub` 中的两个成员, `struct device *intfdev`, `struct usb_device *hdev`, 干什么用的想必不用多说了吧, 第一个, 不管你是 USB 设备也好, PCI 设备也好, SCSI 设备也好, Linux 内核中都为你准备一个 `struct device` 结构体来描述, 所以 `intfdev` 就是和 Hub 相关联的 `struct device` 指针; 第二个, 不管是 Hub 也好, U 盘也好, 移动硬盘也好, USB 鼠标也好, USB Core 都准备一个 `struct usb_device` 来描述, 所以 `hdev` 将是与这个 Hub 相对应的 `struct usb_device` 指针。

而这些在我们调用 `hub_probe` 之前就已经建立好了, 都在参数 `struct usb_interface *intf` 中, 具体怎么得到的, 对于 Root Hub 来说, 这涉及主机控制器的驱动程序, 现在先忽略。但对于一个普通的外接的 Hub, 后面会看到如何得到它的 `struct usb_interface`, 因为建立并初始化一个 USB 设备的 `struct usb_interface` 正是 Hub 驱动里做的事情, 其实也就是我们对 Hub 驱动最好奇的地方。因为找到了这个问题的答案, 我们就知道了对于一个 USB 设备驱动, 其 `probe` 指针是在什么情况下被调用的, 比如这里的 `hub_probe` 对于普通 Hub 来说是谁调用的? 比如之前的 `usb-storage` 中函数 `storage_probe()` 究竟是谁调用的? 这正是我们想知道的。

---

## 8. 蝴蝶效应

朋友, 你相信一只蝴蝶在北京拍拍翅膀, 将使得纽约在几个月后出现比狂风还厉害的龙卷风吗? 看过那部经典的影片《蝴蝶效应》的朋友们一定会说, 这不就是“蝴蝶效应”吗。没错, 蝴蝶效应其实是混沌学理论中的一个概念。它是指对初始条件敏感性的一种依赖现象。蝴蝶效应的原因在于蝴蝶翅膀的运动, 导致其身边的空气系统发生变化, 并引起微弱气流的产生, 而微弱气流的产生又会引起它四周空气或其他系统产生相应的变化, 由此引起连锁反应, 最终导致其他系统的极大变化。

自从 1979 年 12 月麻省理工的洛仑兹在美国科学促进会上作了关于蝴蝶效应的报告之后, 从此蝴蝶效应很快风靡全球, 其迷人的美学色彩和深刻的科学内涵令许多人着迷、激动, 同时发人深省。蝴蝶效应被引入了各个领域, 比如军事、政治、经济, 再后来也被引入到了企业管理。

当然 Linux 中也不会放过如此有哲学魅力的理论。从本质上来说, 蝴蝶效应给人一种对未来行为不可预测的危机感。而 Linux 内核代码中这种感觉更是强烈, 几乎到了无处不在的程度。很多函数, 特别是那种做初始化的函数, 你根本就不知道它在干什么, 只有当你在未来的某个时刻, 看到了另一个函数, 你才会回过头来看, 原来当初是这个函数设置了初始条件。假如你改变了初始条件, 那么后来在某个地方的某个函数的某个行为就会发生改变。但问题是, 你并

不知道这个行为将在什么时候发生。

是不是觉得很抽象？那好，我们来具体讲解一下，比如 935 行，INIT\_DELAYED\_WORK()，这是一个宏，我们给它传递了两个参数，&hub->leds 和 led\_work。对设备驱动熟悉的人不会觉得 INIT\_DELAYED\_WORK() 很陌生，很早就有这个宏了，只不过从 2.6.20 的内核开始这个宏做了改变，原来这个宏是三个参数，后来改成了两个参数，所以经常在网上看见一些人抱怨说最近某个模块编译失败了，在 make 的时候遇见这么一个错误：

```
error: macro "INIT_DELAYED_WORK" passed 3 arguments, but takes just 2
```

当然更为普遍的看到下面这个错误：

```
error: macro "INIT_WORK" passed 3 arguments, but takes just 2
```

于是就让我们来仔细看一看 INIT\_WORK 和 INIT\_DELAYED\_WORK 这两个宏，其实前者是后者的一个特例，它们涉及工作队列。这两个宏都定义于 include/linux/workqueue.h 中：

```
79 #define INIT_WORK(_work, _func) \
80     do { \
81         (_work)->data = (atomic_long_t) WORK_DATA_INIT(); \
82         INIT_LIST_HEAD(&(_work)->entry); \
83         PREPARE_WORK((_work), (_func)); \
84     } while (0) \
85 \
86 #define INIT_DELAYED_WORK(_work, _func) \
87     do { \
88         INIT_WORK(&(_work)->work, (_func)); \
89         init_timer(&(_work)->timer); \
90     } while (0)
```

总之，关于工作队列，Linux 内核实现了一个内核线程，使用 ps 命令看一下您的进程：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ps -el
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   0    1    0  0  76   0 -   195 -   ?           00:00:02 init
1 S   0    2    1  0 -40  - -    0 migrat ?           00:00:00 migration/0
1 S   0    3    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/0
1 S   0    4    1  0 -40  - -    0 migrat ?           00:00:00 migration/1
1 S   0    5    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/1
1 S   0    6    1  0 -40  - -    0 migrat ?           00:00:00 migration/2
1 S   0    7    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/2
1 S   0    8    1  0 -40  - -    0 migrat ?           00:00:00 migration/3
1 S   0    9    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/3
1 S   0   10    1  0 -40  - -    0 migrat ?           00:00:00 migration/4
1 S   0   11    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/4
1 S   0   12    1  0 -40  - -    0 migrat ?           00:00:00 migration/5
1 S   0   13    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/5
1 S   0   14    1  0 -40  - -    0 migrat ?           00:00:00 migration/6
1 S   0   15    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/6
1 S   0   16    1  0 -40  - -    0 migrat ?           00:00:00 migration/7
1 S   0   17    1  0  94  19 -    0 ksofti ?           00:00:00 ksoftirqd/7
5 S   0   18    1  0  70 -5 -    0 worker ?           00:00:00 events/0
1 S   0   19    1  0  70 -5 -    0 worker ?           00:00:00 events/1
5 S   0   20    1  0  70 -5 -    0 worker ?           00:00:00 events/2
```

```

5 S      0      21      1 0 70 -5 -      0 worker ?      00:00:00 events/3
5 S      0      22      1 0 70 -5 -      0 worker ?      00:00:00 events/4
1 S      0      23      1 0 70 -5 -      0 worker ?      00:00:00 events/5
5 S      0      24      1 0 70 -5 -      0 worker ?      00:00:00 events/6
5 S      0      25      1 0 70 -5 -      0 worker ?      00:00:00 events/7

```

看见最后这几行了吗，events/0 到 events/7，“0”和“7”这些都是处理器的编号，每个处理器对应其中的一个线程。要是您的计算机只有一个处理器，那么您只能看到一个这样的线程，events/0，您要是双处理器那您就会看到多出一个 events/1 的线程。我的是 Dell PowerEdge 2950 的机器，有 8 个处理器，所以就是 events/0 到 events/7 了。

那么究竟这些 events 代表什么意思呢？或者说它们具体干什么用的？这些 events 被叫做工作者线程，或者说 Worker Threads，更确切地说，这些应该是默认的工作者线程。而与工作者线程相关的一个概念就是工作队列，或者叫 Work Queue。

工作队列的作用就是把工作推后，交由一个内核线程去执行，更直接地说就是如果你写了一个函数，而你现在不想马上执行它，想在将来某个时刻去执行它，那你可以使用工作队列。中断也是这样，提供一个中断服务函数，在发生中断时去执行，和中断相比，工作队列最大的好处就是可以调度，可以睡眠，灵活性更好。

就比如这里，如果我们将来在某个时刻希望能够调用 led\_work() 这个我们自己写的函数，那么我们所要做的就是利用工作队列。如何利用呢？第一步就是使用 INIT\_WORK() 或者 INIT\_DELAYED\_WORK() 来初始化这么个工作，或者叫任务，初始化了之后，将来如果希望调用这个 led\_work() 函数，那么只要用 schedule\_work() 或者 schedule\_delayed\_work() 就可以了。特别是这里使用的是 INIT\_DELAYED\_WORK()，那么之后就会调用 schedule\_delayed\_work()，这俩是一对函数。它表示，您希望经过一段延时然后再执行某个函数，所以，今后会见到 schedule\_delayed\_work() 这个函数的，而它所需要的参数，一个就是这里的 &hub->leds，另一个就是具体自己需要的延时。&hub->leds 是什么呢？struct usb\_hub 中的成员，struct delayed\_work leds，专门用于延时工作的，再看 struct delayed\_work，这个结构体定义于 include/linux/workqueue.h:

```

35 struct delayed_work {
36     struct work_struct work;
37     struct timer_list timer;
38 };

```

其实就是一个 struct work\_struct 和一个 timer\_list，前者是为了往工作队列里加入自己的工作，后者是为了能够实现延时执行，说得更明白一点，那些 events 线程对应一个结构体，即 struct workqueue\_struct，也就是说它们维护着一个队列，要是你想利用工作队列这个机制呢，可以自己创建一个队列，也可以直接使用 events 对应的这个队列，对于大多数情况来说，都是选择了 events 对应的这个队列，也就是说大家都共用这么一个队列。怎么用呢？先初始化，比如调用 INIT\_DELAYED\_WORK()，这么一初始化，实际上就是为一个 struct work\_struct 结构体绑定一个函数，就比如这里的两个参数，&hub->leds 和 led\_work() 的关系，就最终让 hub\_leds 的 struct

`work_struct` 结构体和函数 `led_work()` 相绑定了起来。它们是怎么绑定的？`struct work_struct` 也是定义于 `include/linux/workqueue.h`：

```
24 struct work_struct {
25     atomic_long_t data;
26 #define WORK_STRUCT_PENDING 0    /* T if work item pending execution */
27 #define WORK_STRUCT_FLAG_MASK (3UL)
28 #define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
29     struct list_head entry;
30     work_func_t func;
31 };
```

看见最后这个成员 `func` 了吗，初始化的目的就是让 `func` 指向 `led_work()`，这就是绑定，所以以后调用 `schedule_delayed_work()` 时，只要传递 `struct work_struct` 的结构体参数即可，不用再每次都把 `led_work()` 这个函数名也给传递一次。

你大概还有一个疑问，为什么只要这里初始化好了，到时调用 `schedule_delayed_work()` 就可以了呢？事实上，`events` 这个线程其实和 `Hub` 的内核线程一样，有事情就处理，没事情就睡眠，也是一个死循环。而 `schedule_delayed_work()` 的作用就是唤醒这个线程，确切地说，是先把自己的这个 `struct work_struct` 插入 `workqueue_struct` 这个队列里，然后唤醒睡眠中的 `events`。然后 `events` 就会去处理，要是有时延，那么它就给您安排延时而后执行，要是没有延时而，或者您设了延时而为 0，那好，那就赶快执行。

这里讲了两个宏，一个 `INIT_WORK()`，一个 `INIT_DELAYED_WORK()`，后者就是专门用于可以有延时的宏，而前者就是没有延时的宏，这里调用的是 `INIT_DELAYED_WORK()`，不过过一会你会看见 `INIT_WORK()` 也被调用了，因为 `Hub` 驱动中还有另一个地方也想利用工作队列这么一个机制，而它不需要延时而，所以就使用 `INIT_WORK()` 进行初始化，然后在需要调用相关函数时调用 `schedule_work()` 即可。

这一节介绍了 Linux 内核中工作队列机制提供的接口，有两对函数：`INIT_DELAYED_WORK()` 对 `schedule_delayed_work()`，`INIT_WORK()` 对 `schedule_work()`。

关于工作队列机制，还会用到另外两个函数，它们是 `cancel_delayed_work(struct delayed_work *work)` 和 `flush_scheduled_work()`。其中 `cancel_delayed_work()` 的意思不言自明，对一个延迟执行的工作来说，这个函数的作用是在这个工作还未执行时就把它给取消。

而 `flush_scheduled_work()` 的作用是为了防止有竞争条件的出现，你要是对竞争条件不是很明白，那也不要紧，反正基本上每次调用 `cancel_delayed_work` 之后你都得调用 `flush_scheduled_work()` 这个函数，特别是对于内核模块，如果一个模块使用了工作队列机制，并且利用了 `events` 这个默认队列，那么在卸载这个模块之前，必须得调用这个函数，这叫做刷新一个工作队列，也就是说，函数会一直等待，直到队列中所有对象都被执行以后才返回。当然，在等待的过程中，这个函数可以进入睡眠。刷新完了之后，这个函数会被唤醒，然后它就返回了。

关于这里的竞争，可以这样理解，events 对应的这个队列本来是按部就班的执行，要是你突然把你的模块给卸载了，或者说把你的那个工作从工作队列里取出来了，那 events 作为队列管理者，它可能根本就不知道，比如说它先想好了，下午 3 点执行队列里的第  $N$  个成员，可是您突然把第  $N-1$  个成员给取走了，那肯定得出错？所以，为了防止出现这种错误，提供了一个 `flush_scheduled_work()` 函数供调用，以消除所谓的竞争条件，其实说竞争太专业了点，说直白一点就是防止混乱吧。

关于这些接口就讲到这里，以后自然会在 Hub 驱动里见到这些接口函数是如何被使用的，到那时候再来看，这就是蝴蝶效应。当我们看到 `INIT_WORK/INIT_DELAYED_WORK()` 时，我们是没法预测未来会发生什么的。所以我们只能拭目以待。

## 9. While You Were Sleeping (一)

继续沿着 `hub_probe()` 往下走，937 行到 941 行，937 行我们在 `usb-storage` 里已然见过了，`usb_set_intfdata(intf, hub)` 的作用就是让 `intf` 和 `hub` 关联起来，从此以后，我们知道 `struct usb_interface *intf`，就可以追溯到与之关联的 `struct usb_hub` 指针。这种思想是很简单的，但也是很重要的，这就好比在网络时代的我们，应该熟练掌握以 Google 为代表的搜索引擎的使用方法。

938 行，设置 `intf` 的 `need_remote_wakeup` 为 1。

940 行，如果这个设备（确切地说是这个 Hub）是高速设备，那么让 `highspeed_hubs` 的值加 1。`highspeed_hubs` 是 `drivers/usb/core/hub.c` 中的全局变量，其定义是这样的：

```
848 static unsigned highspeed_hubs;
```

`static`，静态变量，其实就是 `hub.c` 这个文件中的全局。至于这几个变量有什么用，暂时先不管，用到了再说。

943 行到 947 行，结束了这几行的话，`hub_probe` 就算完了。我们先不用细看每个函数，很显然，`hub_configure` 这个函数是用来配置 Hub 的，返回值小于 0 就算出错了，这里的做法是，如果没有出错那么 `hub_probe` 就返回 0，否则，那就执行 `hub_disconnect()`，断开并且返回错误代码 `-ENODEV`。`hub_disconnect` 函数其实就是和 `hub_probe()` 对应的函数，其关系就像当初 `storage_probe()` 和 `storage_disconnect` 的一样。我们先来看 `hub_configure()`。这个函数又是一个 200 多行的函数。同样还是来自 `drivers/usb/core/hub.c`：

```
595 static int hub_configure(struct usb_hub *hub,
596                         struct usb_endpoint_descriptor *endpoint)
597 {
```

```

598     struct usb_device *hdev = hub->hdev;
599     struct device *hub_dev = hub->intfdev;
600     u16 hubstatus, hubchange;
601     u16 wHubCharacteristics;
602     unsigned int pipe;
603     int maxp, ret;
604     char *message;
605
606     hub->buffer=usb_buffer_alloc(hdev,sizeof(*hub->buffer),GFP_KERNEL,
607                                &hub->buffer_dma);
608     if (!hub->buffer) {
609         message = "can't allocate hub irq buffer";
610         ret = -ENOMEM;
611         goto fail;
612     }
613
614     hub->status = kmalloc(sizeof(*hub->status), GFP_KERNEL);
615     if (!hub->status) {
616         message = "can't kmalloc hub status buffer";
617         ret = -ENOMEM;
618         goto fail;
619     }
620     mutex_init(&hub->status_mutex);
621
622     hub->descriptor = kmalloc(sizeof(*hub->descriptor), GFP_KERNEL);
623     if (!hub->descriptor) {
624         message = "can't kmalloc hub descriptor";
625         ret = -ENOMEM;
626         goto fail;
627     }
628
629     /* Request the entire hub descriptor.
630      * hub->descriptor can handle USB_MAXCHILDREN ports,
631      * but the hub can/will return fewer Bytes here.
632      */
633     ret = get_hub_descriptor(hdev, hub->descriptor,
634                             sizeof(*hub->descriptor));
635     if (ret < 0) {
636         message = "can't read hub descriptor";
637         goto fail;
638     } else if (hub->descriptor->bNbrPorts > USB_MAXCHILDREN) {
639         message = "hub has too many ports!";
640         ret = -ENODEV;
641         goto fail;
642     }
643
644     hdev->maxchild = hub->descriptor->bNbrPorts;
645     dev_info (hub_dev, "%d port%s detected\n", hdev->maxchild,
646              (hdev->maxchild == 1) ? "" : "s");
647
648     wHubCharacteristics =
649         le16_to_cpu(hub->descriptor->wHubCharacteristics);
650
651     if (wHubCharacteristics & HUB_CHAR_COMPOUND) {
652         int i;
653         char portstr [USB_MAXCHILDREN + 1];
654
655         for (i = 0; i < hdev->maxchild; i++)
656             portstr[i] = hub->descriptor->DeviceRemovable

```

```

656             [((i + 1) / 8)] & (1 << ((i + 1) % 8))
657             ? 'F' : 'R';
658     portstr[hdev->maxchild] = 0;
659     dev_dbg(hub_dev, "compound device; port removable status: %s\n",
              portstr);
660 } else
661     dev_dbg(hub_dev, "standalone hub\n");
662
663 switch (wHubCharacteristics & HUB_CHAR_LPSM) {
664 case 0x00:
665     dev_dbg(hub_dev, "ganged power switching\n");
666     break;
667 case 0x01:
668     dev_dbg(hub_dev, "individual port power switching\n");
669     break;
670 case 0x02:
671 case 0x03:
672     dev_dbg(hub_dev, "no power switching (usb 1.0)\n");
673     break;
674 }
675
676 switch (wHubCharacteristics & HUB_CHAR_OCPM) {
677 case 0x00:
678     dev_dbg(hub_dev, "global over-current protection\n");
679     break;
680 case 0x08:
681     dev_dbg(hub_dev, "individual port over-current protection\n");
682     break;
683 case 0x10:
684 case 0x18:
685     dev_dbg(hub_dev, "no over-current protection\n");
686     break;
687 }
688
689 spin_lock_init (&hub->tt.lock);
690 INIT_LIST_HEAD (&hub->tt.clear_list);
691 INIT_WORK (&hub->tt.kevent, hub_tt_kevent);
692 switch (hdev->descriptor.bDeviceProtocol) {
693 case 0:
694     break;
695 case 1:
696     dev_dbg(hub_dev, "Single TT\n");
697     hub->tt.hub = hdev;
698     break;
699 case 2:
700     ret = usb_set_interface(hdev, 0, 1);
701     if (ret == 0) {
702         dev_dbg(hub_dev, "TT per port\n");
703         hub->tt.multi = 1;
704     } else
705         dev_err(hub_dev, "Using single TT (err %d)\n",
706                 ret);
707     hub->tt.hub = hdev;
708     break;
709 default:
710     dev_dbg(hub_dev, "Unrecognized hub protocol %d\n",
711             hdev->descriptor.bDeviceProtocol);
712     break;
713 }

```

```

714
715 /* Note 8 FS bit times == (8 bits / 12000000 bps) ~= 666ns */
716 switch (wHubCharacteristics & HUB_CHAR_TTTT) {
717 case HUB_TTTT_8_BITS:
718     if (hdev->descriptor.bDeviceProtocol != 0) {
719         hub->tt.think_time = 666;
720         dev_dbg(hub_dev, "TT requires at most %d "
721                 "FS bit times (%d ns)\n",
722                 8, hub->tt.think_time);
723     }
724     break;
725 case HUB_TTTT_16_BITS:
726     hub->tt.think_time = 666 * 2;
727     dev_dbg(hub_dev, "TT requires at most %d "
728             "FS bit times (%d ns)\n",
729             16, hub->tt.think_time);
730     break;
731 case HUB_TTTT_24_BITS:
732     hub->tt.think_time = 666 * 3;
733     dev_dbg(hub_dev, "TT requires at most %d "
734             "FS bit times (%d ns)\n",
735             24, hub->tt.think_time);
736     break;
737 case HUB_TTTT_32_BITS:
738     hub->tt.think_time = 666 * 4;
739     dev_dbg(hub_dev, "TT requires at most %d "
740             "FS bit times (%d ns)\n",
741             32, hub->tt.think_time);
742     break;
743 }
744
745 /* probe() zeroes hub->indicator[] */
746 if (wHubCharacteristics & HUB_CHAR_PORTIND) {
747     hub->has_indicators = 1;
748     dev_dbg(hub_dev, "Port indicators are supported\n");
749 }
750
751 dev_dbg(hub_dev, "power on to power good time: %d ms\n",
752         hub->descriptor->bPwrOn2PwrGood * 2);
753
754 /* power budgeting mostly matters with bus-powered hubs,
755  * and battery-powered root hubs (may provide just 8 mA).
756  */
757 ret = usb_get_status(hdev, USB_RECIP_DEVICE, 0, &hubstatus);
758 if (ret < 2) {
759     message = "can't get hub status";
760     goto fail;
761 }
762 le16_to_cpus(&hubstatus);
763 if (hdev == hdev->bus->root_hub) {
764     if (hdev->bus_mA == 0 || hdev->bus_mA >= 500)
765         hub->mA_per_port = 500;
766     else {
767         hub->mA_per_port = hdev->bus_mA;
768         hub->limited_power = 1;
769     }
770 } else if ((hubstatus & (1 << USB_DEVICE_SELF_POWERED)) == 0) {
771     dev_dbg(hub_dev, "hub controller current requirement: %d mA\n",
772             hub->descriptor->bHubContrCurrent);

```



```

773         hub->limited_power = 1;
774         if (hdev->maxchild > 0) {
775             int remaining = hdev->bus_mA -
776                 hub->descriptor->bHubContrCurrent;
777
778             if (remaining < hdev->maxchild * 100)
779                 dev_warn(hub_dev,
780                     "insufficient power available "
781                     "to use all downstream ports\n");
782             hub->mA_per_port = 100;          /* 7.2.1.1 */
783         }
784     } else {          /* Self-powered external hub */
785         /* FIXME: What about battery-powered external hubs that
786          * provide less current per port? */
787         hub->mA_per_port = 500;
788     }
789     if (hub->mA_per_port < 500)
790         dev_dbg(hub_dev, "%u mA bus power budget for each child\n",
791             hub->mA_per_port);
792
793     ret = hub_hub_status(hub, &hubstatus, &hubchange);
794     if (ret < 0) {
795         message = "can't get hub status";
796         goto fail;
797     }
798
799     /* local power status reports aren't always correct */
800     if (hdev->actconfig->desc.bmAttributes & USB_CONFIG_ATT_SELFPOWER)
801         dev_dbg(hub_dev, "local power source is %s\n",
802             (hubstatus & HUB_STATUS_LOCAL_POWER)
803             ? "lost (inactive)" : "good");
804
805     if ((wHubCharacteristics & HUB_CHAR_OCPM) == 0)
806         dev_dbg(hub_dev, "%s over-current condition exists\n",
807             (hubstatus & HUB_STATUS_OVERCURRENT) ? "" : "no ");
808
809     /* set up the interrupt endpoint
810     * We use the EP's maxpacket size instead of (PORTS+1+7)/8
811     * Bytes as USB2.0[11.12.3] says because some hubs are known
812     * to send more data (and thus cause overflow). For root hubs,
813     * maxpktsize is defined in hcd.c's fake endpoint descriptors
814     * to be big enough for at least USB_MAXCHILDREN ports. */
815     pipe = usb_rcvintpipe(hdev, endpoint->bEndpointAddress);
816     maxp = usb_maxpacket(hdev, pipe, usb_pipeout(pipe));
817
818     if (maxp > sizeof(*hub->buffer))
819         maxp = sizeof(*hub->buffer);
820
821     hub->urb = usb_alloc_urb(0, GFP_KERNEL);
822     if (!hub->urb) {
823         message = "couldn't allocate interrupt urb";
824         ret = -ENOMEM;
825         goto fail;
826     }
827
828     usb_fill_int_urb(hub->urb, hdev, pipe, *hub->buffer, maxp, hub_irq,
829         hub, endpoint->bInterval);
830     hub->urb->transfer_dma = hub->buffer_dma;
831     hub->urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

```

```

832
833     /* maybe cycle the hub leds */
834     if (hub->has_indicators && blinkenlights)
835         hub->indicator [0] = INDICATOR_CYCLE;
836
837     hub_power_on(hub);
838     hub_activate(hub);
839     return 0;
840
841 fail:
842     dev_err (hub_dev, "config failed, %s (err %d)\n",
843             message, ret);
844     /* hub_disconnect() frees urb and descriptor */
845     return ret;
846 }

```

不过这个函数虽然长，但是逻辑非常简单，无非就是对 Hub 进行必要的配置，然后就启动 Hub，这个函数的关键就是调用了另外几个经典的函数。下面我们一点一点地来看。

## 10. While You Were Sleeping ( 二 )

老实说，从函数一开始的 598 行直到 627 行都没有什么可说的。其中需要一提的是，606 行，调用 `usb_buffer_alloc()` 申请内存，赋给 `hub->buffer`。614 行，调用 `kmalloc()` 申请内存，赋给 `hub->status`。622 行，调用 `kmalloc()` 申请内存，赋给 `hub->descriptor`。当然也别忘了这中间的某行，初始化一把互斥锁，`hub->status_mutex`。以后会用得着的，到时候我们就会看到 `mutex_lock/mutex_unlock()` 这对函数来获得互斥锁/释放互斥锁。不过这把锁用得不多，总共也就两个函数中调用了。

633 行，`get_hub_descriptor()`。这一行带领我们步入了 Hub 协议。从此摆在我们面前的将不仅仅是 USB 协议本身了，又加了另一座大山，那就是 Hub 协议。

其实 Hub 协议也算 USB 协议，但是由于 Hub 作为一种特殊的 USB 设备，为了简明起见，下面我将把这一章称做 Hub 协议。而接下来我们的一言一行，都必须遵守 Hub 协议了。先看 `get_hub_descriptor`，这就是发送一个请求，或者说发送一个控制传输的控制请求，以获得 Hub 的描述符。基本上 Hub 驱动的这些函数，大多都是来自 `drivers/usb/core/hub.c` 中：

```

137 /* USB 2.0 spec Section 11.24.4.5 */
138 static int get_hub_descriptor(struct usb_device *hdev, void *data, int size)
139 {
140     int i, ret;
141
142     for (i = 0; i < 3; i++) {
143         ret = usb_control_msg(hdev, usb_rcvctrlpipe(hdev, 0),
144                               USB_REQ_GET_DESCRIPTOR, USB_DIR_IN | USB_RT_HUB,
145                               USB_DT_HUB << 8, 0, data, size,

```

```
146             USB_CTRL_GET_TIMEOUT);
147         if (ret >= (USB_DT_HUB_NONVAR_SIZE + 2))
148             return ret;
149     }
150     return -EINVAL;
151 }
```

这里只是使用 USB Core 里提供的 `usb_control_msg` 函数向 Hub 发送 `GET_DESCRIPTOR` 请求。每一个请求怎么设置都是在 `spec` 里边规定的，对于 `GET_DESCRIPTOR` 这个请求，如图 2.10.1 所示。

bmRequestType	bRequest	wValue	wLength	Data
10000000B	GET_DESCRIPTOR	描述符的类型和序号(index)	描述符长度	描述符

图 2.10.1 GET\_DESCRIPTOR 请求

协议中规定，`bmRequestType` 必须是 `10100000B`，normally，`GET_DESCRIPTOR` 时，`bmRequestType` 应该等于 `10000000B`。D7 为方向位，1 就说明是 Device-to-host，即 IN，D6-5 这两位表示请求的类型，可以是标准的类型，或者是 Class 特定的，或者是 Vendor 特定的，01 就表示 Class 特定的。D4-0 表示接受者，可以是设备，可以是接口，可以是端点。这里为 0 表示接收者是设备。

`USB_DT_HUB` 等于 29，而 Hub 协议规定 `GET_DESCRIPTOR` 这个请求的 `wValue` 就该是描述符的类型和序号，`wValue` 作为一个 word 有 16 位，所以高 8 位放描述符类型，而低 8 位放描述符的序号，`spec 11.24.2.10` 里规定了，Hub 描述符的描述符序号必须为 0。而实际上，`spec 9.4.3` 里也规定了，对于非配置描述符和字符串描述符的其他几种标准描述符，其描述符序号必须为 0。而对于配置描述符和字符串描述符来说，这个描述符序号用来表示它们的编号，比如一个设备可以有多个配置描述符，编号从 0 开始。所以对于 Hub 描述符来说，`wValue` 就是高 8 位表示描述符类型，而低 8 位设成 0 就可以了，这就是为什么“<<8”了。`spec` 中的 Table 9-5 定义了各种描述符的类型，比如设备描述符是 1，配置描述符是 2，而 Hub 这里规定了，它的描述符类型是 29h。

而 `USB_DT_HUB=(USB_TYPE_CLASS|0x09)`，`USB_TYPE_CLASS` 就是 `0x01<<5`。所以这里 `USB_DT_HUB` 就是 `0x29`，即 29h。

`USB_CTRL_GET_TIMEOUT` 是一个宏，值为 5000，即表示 5 秒超时。

`USB_DT_HUB_NONVAR_SIZE` 也是一个宏，值为 7，为什么为 7 呢？首先请你明白，我们这里要获得的是 Hub 描述符，这是只有 Hub 协议才有的一个描述符，就是说对于 Hub 协议来说，除了通常的 USB 设备有的那些设备描述符，接口描述符，端点描述符以外，Hub 协议自

己也定义了一个描述符，这就叫 Hub 描述符。而 Hub 描述符的前 7 个字节是固定的，表示的意思也是确定的，但从第 8 个字节开始，一切就都充满了变数。所以前 7 个字节被称为非变量，即 NON-Variable，而从第 7 个开始以后的被称为变量，即 Variable。这些变量取决于 Hub 上面端口的个数。

所谓的变量不是说字节的内容本身是变化的，而是说描述符具体有几个字节是变化的，比如 Hub 上面有两个端口。那么这个 Hub 的描述符的字节数和 Hub 上面有 4 个端口的情况就是不一样的，显然，有 4 个端口就要记录更多的信息，当然描述符里的内容就多一些，从而描述符的字节长度也不一样。

usb\_control\_msg()如果执行成功，那么返回值将是成功传输的字节长度，对这里来说，就是传输的 Hub 描述符的字节长度。结合 Hub 协议来看，这个长度至少应该是 9，所以这里判断如果大于等于 9，那就返回。当然你要问为什么这里要循环三次，这是为了防止通信错误。Hub 驱动中很多地方都这样做了，主要是因为很多设备在发送它们的描述符时总是出错。所以没有办法了，多试几次吧。

## 11. While You Were Sleeping ( 三 )

get\_hub\_descriptor()结束了，然后就返回 hub\_configure()函数中来。635 行到 642 行，判断刚才的返回值，小于零当然是出错了，大于零也还要多判断一次，USB\_MAXCHILDREN 是自己定义的一个宏，值为 31。查看一下 include/linux/usb.h:

```
324 #define USB_MAXCHILDREN (31)
```

其实 Hub 可以接一共 255 个端口，不过实际上遇到的 Hub 最多的也就是支持 10 个端口。所以 31 个端口基本上够用了。当然你要是心血来潮把这个宏改成 100 或 200，那也不会出错。

我们来看一下 Hub 描述符对应的数据结构。struct usb\_hub 中有一个成员，struct usb\_hub\_descriptor \*descriptor，就是表示 Hub 描述符的，它定义于 drivers/usb/core/hub.h，与 spec 中的 Table 11-13 相对应。

```
130 struct usb_hub_descriptor {
131     __u8  bDescLength;
132     __u8  bDescriptorType;
133     __u8  bNbrPorts;
134     __le16 wHubCharacteristics;
135     __u8  bPwrOn2PwrGood;
136     __u8  bHubContrCurrent;
137     /* add 1 bit for hub status change; round to Bytes */
138     __u8  DeviceRemovable[(USB_MAXCHILDREN + 1 + 7) / 8];
139     __u8  PortPwrCtrlMask[(USB_MAXCHILDREN + 1 + 7) / 8];
```

```
140 } __attribute__((packed));
```

看见了没有，至少 9 个字节吧，接下来我们会用到 `bNbrPorts`，它代表 Number of downstream facing ports that this hub supports，就是说这个 Hub 所支持的下行端口，刚才这里判断的就是这个值是不是比 31 还大，如果是，那么就出错了。

`bHubContrCurrent` 是 Hub 控制器的最大电流需求，`DeviceRemoveable` 是用来判断这个端口连接的设备是否可以移除的，每一个 bit 代表一个端口，如果该 bit 为 0，则说明可以被移除；如果该 bit 为 1，就说明不可以移除。而 `wHubCharacteristics` 就相对来说麻烦一点了，它记录了很多信息，后面有相当一部分的代码都是在判断这个值。

648 行，用一个临时变量 `wHubCharacteristics` 来代替描述符里的 `wHubCharacteristics`。从 650 行就开始判断了，首先判断是不是复合设备。在 `drivers/usb/core/hub.h` 中定义了如下一些宏。

```
99 #define HUB_CHAR_LPSM          0x0003 /* D1 .. D0 */
100 #define HUB_CHAR_COMPOUND      0x0004 /* D2          */
101 #define HUB_CHAR_OCPM          0x0018 /* D4 .. D3 */
102 #define HUB_CHAR_TTTT          0x0060 /* D6 .. D5 */
103 #define HUB_CHAR_PORTIND       0x0080 /* D7          */
```

结合 spec 中的 Table 11-13，意思很明显。650 行到 661 行，如果是复合设备（复合设备就是说这个设备它可能是几种设备绑在一起的，比如既可以当 Hub 用又可以有别的功能）那么就用一个数组 `portstr[]` 来记录每一个端口的设备是否可以被移除。然后打印出调试信息来。如果看不懂，把 `i` 用 0, 1, 2, 3 这些数字代入进去就明白了。

663 行到 674 行，`HUB_CHAR_LPSM`，表示 power switching（电源切换）的方式，不同的 Hub 有不同的 power switching 的方式，`ganged power switching` 指的是所有端口一次性上电。而 USB 1.0 的 Hub 却没有 power switching 这个说法。

676 行到 687 行，`HUB_CHAR_OCPM`，表示过流保护模式，如果不明白也无所谓，这几行无非就是打印一些调试信息。

689 行到 691 行，先是初始化一个自旋锁 `hub->tt.lock`，而是 `struct usb_hub` 中的成员。

```
176 struct usb_tt {
177     struct usb_device      *hub; /* upstream highspeed hub */
178     int                    multi; /* true means one TT per port */
179     unsigned               think_time; /* think time in ns */
180
181     /* for control/bulk error recovery (CLEAR_TT_BUFFER) */
182     spinlock_t             lock;
183     struct list_head       clear_list; /* of usb_tt_clear */
184     struct work_struct     kevent;
185 };
```

知道 `tt` 是什么吗？`tt` 即 transaction translator。你可以把它想成一块特殊的电路，是 Hub 里面的电路，确切地说是高速 Hub 中的电路。

我们知道 USB 设备有三种速度，分别是 Low Speed、Full Speed、High Speed。即所谓的低速、全速、高速，以前只有低速/全速的设备，没有高速的设备。后来才出现了高速的设备，包括主机控制器。以前只有两种接口的设备，OHCI/UHCI，这都是在 USB spec 1.0 时，后来 USB spec 2.0 推出了 EHCI，高速设备应运而生。

Hub 也有高速 Hub 和低速/全速的 Hub，但是这就产生一个兼容性问题了，高速的 Hub 是否能够支持低速/全速的设备呢？一般来说是不支持的，于是有了叫做 tt 的电路，它就负责高速和低速/全速的数据转换。如果一个高速设备中有这个 tt 电路，那么就可以连接低速/全速设备，要不然，低速/全速设备就没法使用，只能连接到 OHCI/UHCI 的 Hub 口里。

690 行，初始化一个队列，hub->tt.clear\_list。691 行，在这里我们看到了 INIT\_WORK()，hub->tt.kevent 是一个 struct work\_struct 的结构体，而 hub\_tt\_kevent 是我们定义的函数，将会在未来某个时间去执行。另外，tt 有两种，一种是 single tt，另一种是 multi tt。前者表示整个 Hub 就是一个 tt，而 multi tt 表示每个端口都配了一个 tt。大多数 Hub 是 single tt，因为一个 Hub 有一个 tt 就够了。

692 行的 switch，hdev->descriptor.bDeviceProtocol，别看走眼了，刚才咱们一直是判断 hub->descriptor，而这里是 hdev->descriptor，hdev 是 struct usb\_device 结构体指针，一进入这个 hub\_configure() 函数就赋了值的，其实就是和这个 Hub 相关的 struct usb\_device 指针。所以这里判断的描述符是标准的 USB 设备描述符，而其中 bDeviceProtocol 的含义在 hub 协议中有专门的规定。

这一段就是为了设置 tt，对照 Hub 协议可知，低速/全速的 Hub 的 bDeviceProtocol 是 0，这种 Hub 就没有 tt。所以直接 break，什么也不用设置。对于高速的 Hub，其 bDeviceProtocol 为 1 表示是 single tt 的；bDeviceProtocol 为 2 表示是 multiple tt 的。

对于 case 2，这里调用了 usb\_set\_interface，根据 spec 中的 11.23.1 小节，对于低速/全速的 Hub，其设备描述符中的 bDeviceProtocol 为 0，而接口描述符中的 bInterfaceProtocol 也为 0。而对于高速的 Hub，其中 single tt 的 Hub 其设备描述符中的 bDeviceProtocol 是 1，而接口描述符的 bInterfaceProtocol 则是 0。然而，multiple tt 的 Hub 另外还有一个接口描述符，以及相应的一个端点描述符，它的设备描述符的 bDeviceProtocol 必须设置成 2。其第一个接口描述符的 bInterfaceProtocol 为 1，而第二个接口描述符的 bInterfaceProtocol 则是 2。

Hub 只有一个接口，但是可以有两种设置。usb\_set\_interface 就是把这个接口（接口 0）设置 1，因为默认都是设置 0。关于 SET\_INTERFACE 这个请求，是 USB spec 2.0 的一个错误。另外，hub->tt.hub 就是 struct usb\_device 的结构体指针。hub->tt.multi 就是一个 int 型的 flag，设为 1 就表示这是一个 multi tt 的 Hub。

716 行，HUB\_CHAR\_TTTT，后两个 tt 就是 think time 的意思，也就是说 tt 在处理两个低速/全速的交易之间需要一点时间来缓冲，而这个最大的间隔就叫做 tt think time。这个时间当然

不会很长。不过需要注意，这里用的单位是 FS bit time，我们知道 FS 就是 Full Speed，其速度是 12 MB/s，其实也就是 1200 0000Bit/s，8 FS bit time 就是 8Bits / 1200 0000 Bits per second，即约等于 666ns。所以这里就用 666ns 来记录了。不过以后你会发现，其实 think\_time 这个值我们就没用过。

746 行到 749 行， HUB\_CHAR\_PORTIND，这个表示 port indicator。0 说明不支持，1 说明支持。indicator 是干什么用的？indicator 就是 Hub 上面的那个指示灯。通常是两种颜色，绿色和琥珀色。你是不是还经常看见红色？其实什么颜色无所谓，不过 spec 上面是给出的这两种颜色。其实就是一个 LED 灯，提供两种颜色，或者是两个 LED 灯。

其实大多数 Hub 是有指示灯的，不管 USB Hub 还是别的 Hub，或者 Switch，统统有指示灯，因为指示灯对于工程师调试硬件产品是很有帮助的。产品出现了问题，首先查看指示灯也许就知道怎么回事了，我记得以前在家里上网时，网络中断了，打上海电信客服的电话，人家首先就是问我那几个指示灯是如何亮的。

757 行，usb\_get\_status()，来自 drivers/usb/core/message.c:

```
900 int usb_get_status(struct usb_device *dev, int type, int target, void *data)
901 {
902     int ret;
903     u16 *status = kmalloc(sizeof(*status), GFP_KERNEL);
904
905     if (!status)
906         return -ENOMEM;
907
908     ret = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
909                          USB_REQ_GET_STATUS, USB_DIR_IN | type, 0, target, status,
910                          sizeof(*status), USB_CTRL_GET_TIMEOUT);
911
912     *(u16 *)data = *status;
913     kfree(status);
914     return ret;
915 }
```

又是一个控制传输，发送的是一个控制请求，GET\_STATUS 是 USB 的标准请求之一，如图 2.11.1 所示。

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_STATUS	0	0	2	设备，接口，或端点状态
10000001B					
10000010B					

图 2.11.1 GET\_STATUS 请求

这个请求的类型有三种，一种是获得设备的状态，一种是获得接口的状态，另一种是获得端点的状态，这里传递的是 USB\_RECIP\_DEVICE，也就是获得设备的状态。那么函数返回之后，设备的状态就被保存在了 hubstatus 里面了。

763 行至行 788 行，就是一个 if...else 组合。首先 if 函数判断这个设备是不是 Root Hub，如果是 Root Hub，然后判断 hdev->bus\_mA，这个值是在主机控制器的驱动程序中设置的。通常计算机的 USB 端口可以提供 500mA 的电流，不过主机控制器那边有一个成员 power\_budget，在主机控制器的驱动程序中，Root Hub 的 hdev->bus\_mA 被设置为 500mA。即如果你有兴趣看一下 drivers/usb/core/hcd.c，你会注意到在 usb\_add\_hcd 这个函数中有下面这两行：

```
1637      /* starting here, usbcore will pay attention to this root hub */
1638      rhdev->bus_mA = min(500u, hcd->power_budget);
```

power\_budget 是主机控制器自己提供的，它可以是 0，表示没有限制。所以我们这里判断是不是等于 0，或者是不是大于等于 500mA，如果是，那么就设置 hub->mA\_per\_port 为 500mA，mA\_per\_port 就是提供给每一个端口的电流。那么如果说 bus\_mA 是 0 到 500 之间的某个数，那么说明这个 Hub 没法提供达到 500mA 的电流，就是主机控制器那边提供不了这么大的电流，那么 hub->mA\_per\_port 就设置为 hdev->bus\_mA。同时，对于这种主机控制器那边限制了电流的情况，记录下来，hub->limited\_power 这个标志位设置为 1。

那么如果不是 Root Hub，又分两种情况。USB\_DEVICE\_SELF\_POWERED，hubstatus 里的这一位表示这个 Hub 是不是自己供电的，因为外接的 Hub 也有两种供电方式，自己供电或者选择请求总线供电。770 行如果满足条件，那就说明这又是一个要总线供电的 Hub，于是 limited\_power 也设置为 1。774 行，maxchild>0。然后定义了一个 int 变量 remaining 来记录剩下多少电流，hdev->bus\_mA 就是这个 Hub（不是 Root Hub）上行口的电流，而 bHubContrCurrent 在前面说过了，就是 Hub 需要的电流。两者相减就是剩下的。但是在 USB 端口上，最小的电流负载就是 100mA，这个叫做单元负载（unit load）。

778 行的意思很显然，比如你这个 Hub 有 4 个端口，即 maxchild 为 4，那么你最起码你得剩一个 400mA 电流的端口吧，因为如果某个端口电流小于 100mA 的话，设备是没法正常工作的。然后，782 行，警告归警告，最终还是设置 mA\_per\_port 为 100mA。

784 行，如果是自己供电的那种 Hub，那没得说，直接设置为 500mA 吧。

793 行，hub\_hub\_status()，这个函数还是来自 drivers/usb/core/hub.c：

```
531 static int hub_hub_status(struct usb_hub *hub,
532                          u16 *status, u16 *change)
533 {
534     int ret;
535
536     mutex_lock(&hub->status_mutex);
537     ret = get_hub_status(hub->hdev, &hub->status->hub);
538     if (ret < 0)
539         dev_err (hub->intfdev,
540                 "%s failed (err = %d)\n", __FUNCTION__, ret);
541     else {
542         *status = le16_to_cpu(hub->status->hub.wHubStatus);
543         *change = le16_to_cpu(hub->status->hub.wHubChange);
544         ret = 0;
```



```

545     }
546     mutex_unlock(&hub->status_mutex);
547     return ret;
548 }

```

和刚才那个 `get_hub_status` 不一样的是，刚才那个 `GET_STATUS` 是标准的 USB 设备请求，每个设备都会有的，但是现在这个请求是 Hub 自己定义的。

最后状态保存在 `status` 和 `change` 里面，即 `hubstatus` 和 `hubchange` 里面。而函数 `hub_hub_status` 的返回值正常就是 0，否则就是负的错误码。

799 行到 807 行都是打印调试信息。

809 行开始就是真正地干正经事儿了。我们知道 `usb-storage` 里面最常见的传输方式就是控制和批量传输，而对于 Hub，它的传输方式就是控制和中断，最有特色的正是它的中断传输。注意咱们在调用 `hub_configure` 时传递进来的第二个参数是 `endpoint`，前面我们已经说了，这正代表着 Hub 的中断端点，所以 815 行应该一眼就能看出这一行就是获得连接主机与这个端点的这条管道，是一条中断传输的管道。不过请注意，Hub 里面的中断端点一定是 IN 而不是 OUT 的，spec 就这么规定的。

816 行，`usb_maxpacket` 我们是第一次遇见，其实它就是获得一个端点描述符里面的 `wMaxPacketSize`，赋给 `maxp`，一个端点一次最多传输的数据就是 `wMaxPacketSize`。不可以超过它。我们前面为 `hub->buffer` 申请了内存，这里 `maxp` 如果大于这个 `size`，那么不可以，就让它等于 `hub->buffer` 的 `size`。

821 行，申请一个 `urb`，然后填充一个 `urb`，可以对照 `usb_fill_int_urb()` 的代码看一下这里的 `urb` 内容是什么。`usb_fill_int_urb()` 来自 `include/linux/usb.h`：

```

1242 static inline void usb_fill_int_urb (struct urb *urb,
1243                                     struct usb_device *dev,
1244                                     unsigned int pipe,
1245                                     void *transfer_buffer,
1246                                     int buffer_length,
1247                                     usb_complete_t complete_fn,
1248                                     void *context,
1249                                     int interval)
1250 {
1251     spin_lock_init(&urb->lock);
1252     urb->dev = dev;
1253     urb->pipe = pipe;
1254     urb->transfer_buffer = transfer_buffer;
1255     urb->transfer_buffer_length = buffer_length;
1256     urb->complete = complete_fn;
1257     urb->context = context;
1258     if (dev->speed == USB_SPEED_HIGH)
1259         urb->interval = 1 << (interval - 1);
1260     else
1261         urb->interval = interval;
1262     urb->start_frame = -1;

```

对比一下形参和实参，重点关注这么几项，`transfer_buffer` 就是 `hub->buffer`，`transfer_buffer_length` 就是 `maxp`，`complete` 就是 `hub_irq`，`context` 被赋为了 `Hub`，而 `interval` 被赋为 `endpoint->bInterval`。

至于 `interval` 为什么不一样，其实是因为单位不一样，早年，提到 USB 协议，人们会提到 `frame`，即帧，后来出现了一个新的名词，叫做微帧，即 `microframe`。一个帧是 1 毫秒，而一个微帧是八分之一毫秒，也就是 125 微秒。要知道我们刚才说了，这里传递进来的 `interval` 实际上是 `endpoint->bInterval`，要深刻认识这段代码背后的哲学意义，需要知道 USB 中断传输究竟是怎么进行的。

## 12. While You Were Sleeping ( 四 )

我们说过，`Hub` 里面的中断端点是 `IN`，不是 `OUT`。但这并不说明凡是中断传输数据一定是从设备到主机。不过 `Hub` 需要的确实只是 `IN` 的传输。

首先，中断是由设备产生的。在 USB 的世界里两个重要角色，主机和设备。

那么什么是 `interval` 呢？`interval` 就是间隔期，什么叫间隔期？首先你要明白，中断传输绝不是一种周期性的传输。那为什么还要有一个间隔期呢？实际上是这样的，尽管中断本身不会定期发生，但是有一个事情是周期性的，对于 `IN` 的中断端点，主机会定期向设备访问。

那么具体来讲，在 USB 的世界里，体现的是一种设计者们美好的愿望。也就是说，设计者们眼看着现实世界中的公仆们都很让人失望，所以在设计 `spec` 时是这么规定的，主机必须要体恤民情，而且，很重要的一点，端点可以提出自己的愿望，比如希望主机多久来探望一次，一个端点的端点描述符里 `bInterval` 这么一项写得很清楚，就是它渴望的总线访问周期，或者说它期望主机隔多久就能看望一下它。

设备要进行中断传输，需要提交一个 `urb`，里面注明这个探访周期，而主机答应不答应，在主机控制器的驱动程序里才知道，我们不管，主机当然有一个评判标准，中断传输可以用在低速/全速/高速设备中，而高速传输可以接受在每个微帧有多达 80% 的时间是进行定期传输，（包括中断传输和等时传输），而全速/低速传输则可以接受每个帧最多有 90% 的时间来进行定期传输。所以，主机会统一来安排，它会听取每一个群众（设备）的意见或者说愿望，然后它来统一安排，日程安排得过来就给安排，安排不过来那么就返回错误。这些都是主机控制器驱动做的。那么如果主机同意，或者说主机控制器接受了群众的意见，比如告诉它希望它每周来看端点一次，那好，它每周来一次，问断点有没有什么困难（中断）。

从技术的角度来讲，主机控制器定期给你发送一个 IN token，就是说发送这么一个包，而你如果有中断等在那里，那你就告诉它，你有了中断。同时你会把与这个中断相关的数据发送给主机，这就是中断传输的数据阶段，显然，这就是 IN 方向的传输。然后主机接收到数据之后它会发送回来一个包，向你确认一下。

那么如果主机控制器发送给你一个 IN token 时，你没有中断，那怎么办呢？你还是要回应一声，说没有中断。当然还有另一种情况是，你可能人不在家，出差去了，那么你可以在家留一个便条，告诉人家你没法回答。以上三种情况，对应专业术语来说就是，第一种，你回应的是 DATA，主机回应的是 ACK 或者是 Error。第二种，你回应的是 NAK，第三种，你回应的是 STALL。

顺便解释一下 OUT 类型的中断端点是如何进行数据传输（虽然 Hub 里根本就没有这种款式的端点）。分三步走，第一步，主机发送一个 OUT token 包，然后第二步就直接发送数据包，第三步，设备回应，也许回应 ACK，表示成功接收到了数据；也许回应 NAK，表示失败了；也许回应 STALL，表示设备端点本身有问题，传输没法进行。

现在可以回到刚才那个 interval 问题了，因为不同速度的 interval 的单位不一样，所以同样一个数字表达的意思也不一样。那么对于高速设备来说，比如它的端点的 bInterval 的值为  $n$ ，那么这表示它渴望的周期是  $2^{n-1}$  个微帧，比如  $n$  为 4，那么就表示  $2^3$  个微帧，即 8 个 125 微秒，也就是 1 毫秒。对于高速设备来说，spec 里规定， $n$  的取值必须在 1 到 16 之间，而对于全速设备来说，其渴望周期在 spec 里有规定，必须是在 1 毫秒到 255 毫秒之间，对于低速设备来说，其渴望周期必须在 10 毫秒到 255 毫秒之间。可见，对于全速/低速设备来说，不存在这种指数关系，所以 `urb->interval` 直接被赋值为 bInterval，而高速设备由于这种指数关系，bInterval 的含义就不是那么直接，而是表示那个幂指数。而 `start_frame` 是专门给等时传输用的，所以我们不用管了，这里当然直接设置为 -1 即可。

终于，我们明白了这个中断传输，明白了这个 `usb_fill_int_urb()` 函数，于是我们再次回到 `hub_configure()` 函数中来，830 行和 831 行，这个没什么好说的，`usb-storage` 里面也就这么设置的，能用 DMA 传输当然要用 DMA 传输。

834 行，`has_indicators` 不用说了，刚刚才介绍的，有就设置为了 1，没有就是 0，不过 Hub 驱动里提供了一个参数，叫做 `blinkerlights`，指示灯有两种特性，一个是亮，一个是闪，有灯了，就会亮了，但是不一定会闪，所以 `blinkerlights` 就表示灯闪不闪，这个参数可以在我们加载模块时设置，默认值是 0，在 `drivers/usb/core/hub.c` 中有定义：

```
91 static int blinkerlights = 0;
92 module_param (blinkerlights, bool, S_IRUGO);
93 MODULE_PARM_DESC (blinkerlights, "true to cycle leds on hubs");
```

以上都是和模块参数有关的。如果这两个条件都满足，就设置 `hub->indicator [0]` 为 `INDICATOR_CYCLE`。

837 行, `hub_power_on()`, 这个函数的意图就是相当于打开电视机开关。

838 行, `hub_activate()`。在讲这个函数之前, 先看一下 `hub_configure()` 中剩下的最后几行, `hub_activate()` 之后就返回了。841 行的 `fail` 是行标, 之前那些出错的地方都有 `goto fail` 语句跳转过来, 而且错误码也记录在了 `ret` 里面, 于是返回 `ret`。好, 让我们来看一下 `hub_activate()`。这个函数不长, 依然来自 `drivers/usb/core/hub.c`:

```
514 static void hub_activate(struct usb_hub *hub)
515 {
516     int      status;
517
518     hub->quiescing = 0;
519     hub->activating = 1;
520
521     status = usb_submit_urb(hub->urb, GFP_NOIO);
522     if (status < 0)
523         dev_err(hub->intfdev, "activate --> %d\n", status);
524     if (hub->has_indicators && blinkenlights)
525         schedule_delayed_work(&hub->leds, LED_CYCLE_PERIOD);
526
527     /* scan all ports ASAP */
528     kick_khubd(hub);
529 }
```

`quiescing` 和 `activating` 就是两个标志符。`activating` 这个标志的意思不言自明, 而 `quiescing` 这个标志的意思就容易让人疑惑了。`quiescing` 有停顿, 停止, 停息的意思, 怎么一看 `activating` 和 `quiescing` 就是一对反义词, 可是如果真的是起相反的作用, 那么一个变量不就够了吗? 可以为 1, 可以为 0, 不就表达了两种意思了吗?

512 行, 我们太熟悉了。前面我们调用 `usb_fill_int_urb()` 填充好了一个 `urb`, 这会儿就该提交了, 然后主机控制器就知道了, 然后如果一切顺利的话, 主机控制器就会定期来询问 Hub, 问它有没有中断, 有的话就进行中断传输, 这个我们在前面讲过了。

524 行, 又和刚才一样的判断, 不过这次判断条件满足了以后就会执行一个函数, `schedule_delayed_work()`, 终于看到这个函数被调用了, 前面分析清楚了, 这里一看我们就知道, 延时调用, 调用的是 `leds` 对应的 `work` 函数, 即我们当初注册的 `led_work()`。这里 `LED_CYCLE_PERIOD` 就是一个宏, 表明延时多久, 这个宏在 `drivers/usb/core/hub.c` 中定义好了:

```
208 #define LED_CYCLE_PERIOD      ((2*HZ)/3)
```

关于这个指示灯的代码我们以后再分析, 我们真正需要花时间关注的是 Hub 作为一个特殊的 USB 设备它是如何起到连接主机和设备的作用。

继续看, 528 行, `kick_khubd(hub)`, 来自 `drivers/usb/core/hub.c`:

```
316 static void kick_khubd(struct usb_hub *hub)
317 {
```

```

318     unsigned long   flags;
319
320     /* Suppress autosuspend until khubd runs */
321     to_usb_interface(hub->intfdev)->pm_usage_cnt = 1;
322
323     spin_lock_irqsave(&hub_event_lock, flags);
324     if (list_empty(&hub->event_list)) {
325         list_add_tail(&hub->event_list, &hub_event_list);
326         wake_up(&khubd_wait);
327     }
328     spin_unlock_irqrestore(&hub_event_lock, flags);
329 }

```

这才是我们真正期待的一个函数，看见 `wake_up()` 函数，就知道怎么回事了吧。先看 321 行，`int pm_usage_cnt` 是 `struct usb_interface` 的一个成员，`pm` 是电源管理，`usage_cnt` 是使用计数，这里的意图很明显，要使用 `Hub` 了，就别让电源管理把它给挂起来了。还是不明白？用过笔记本电脑吧？我发现很多时候合上笔记本电脑它就会自动进入休眠，可是有时候我会发现合上笔记本电脑以后，笔记本电脑并没有休眠，后来总结出来规律了，在网上下载数据时基本上笔记本电脑是不会进入睡眠的，理由很简单，它发现你有活动着的网络线程，不过，这里，我们计数的目的不是记录网络线程，而是告诉 `USB Core`，我们拒绝自动挂起，具体的处理会由 `USB Core` 来统一操作，`USB Core` 那边自然会判断 `pm_usage_cnt` 的，只要我们这里设置了就可以了。

324 行到 327 行这段 `if` 判断语句，很显然，现在我们是第一次来到这里，不用说，`hub->event_list` 是空的，所以，满足条件，于是 `list_add_tail()` 会被执行，关于队列操作函数，前面也讲过了，所以这里也不用困惑了，就是往那个总的队列 `hub_event_list` 里面加入 `hub->event_list`，然后调用 `wake_up(&khubd_wait)` 去唤醒那个“昏睡”了的 `hub_thread()`。从此 `hub_events()` 函数将再次被执行。

至此为止，整个关于 `Hub` 的配置就讲完了，从现在开始，`Hub` 就可以正式上班了。而我们也终于完成了一个目标，唤醒了 `hub_thread()`，进入 `hub_events()`。

## 13. 再向虎山行

再一次进入 `while` 这个死循环。

第一次来这里时，`hub_event_list` 是空的，可是这一次不是了，我们刚刚在 `kick_khubd()` 里面才执行了往这个队列里插入的操作，所以我们不会再像第一次一样，从 2621 行的 `break` 跳出循环。相反，我们直接走到 2624 行，把刚才插入队列的那个节点取出来，存为 `tmp`，然后把 `tmp` 从队列里删除掉（是从队列里删除，不是把 `tmp` 本身给删除）。

2627 行, `list_entry()`, 这是一个经典的函数, 或者说宏。通过这个宏这里得到的是触发 `hub_events()` 的 Hub。2628 行, 同时用局部变量 `hdev` 记录 `hub->hdev`。2629 行, 又得到对应的 `struct usb_interface` 和 `struct device`。

2640 行, `usb_get_intf()`, 只是一个引用计数, 是 USB Core 提供的一个函数, 以前黑客们推荐用另一个引用计数的函数 `usb_get_dev()`, 但在当今随着一个 USB 设备成为多个接口耦合的情况的出现, `struct usb_device` 实际上已经快淡出历史舞台了, 现在在驱动程序里关注的最多的就是接口, 而不是设备。和 `usb_get_intf()` 对应的另一个函数叫做 `usb_put_intf()`, 很显然, 一个是增加引用计数, 一个减少引用计数。这个函数我们马上就能看到。

前面的 `hub_events()` 只看到 2641 行, 现在继续往下看。

```

2642
2643         /* Lock the device, then check to see if we were
2644          * disconnected while waiting for the lock to succeed. */
2645         if (locktree(hdev) < 0) {
2646             usb_put_intf(intf);
2647             continue;
2648         }
2649         if (hub != usb_get_intfdata(intf))
2650             goto loop;
2651
2652         /* If the hub has died, clean up after it */
2653         if (hdev->state == USB_STATE_NOTATTACHED) {
2654             hub->error = -ENODEV;
2655             hub_pre_reset(intf);
2656             goto loop;
2657         }
2658
2659         /* Autoresume */
2660         ret = usb_autopm_get_interface(intf);
2661         if (ret) {
2662             dev_dbg(hub_dev, "Can't autoresume: %d\n", ret);
2663             goto loop;
2664         }
2665
2666         /* If this is an inactive hub, do nothing */
2667         if (hub->quiescing)
2668             goto loop_autopm;
2669
2670         if (hub->error) {
2671             dev_dbg(hub_dev, "resetting for error %d\n",
2672                     hub->error);
2673
2674             ret = usb_reset_composite_device(hdev, intf);
2675             if (ret) {
2676                 dev_dbg(hub_dev,
2677                         "error resetting hub: %d\n", ret);
2678                 goto loop_autopm;
2679             }
2680
2681             hub->nerrors = 0;
2682             hub->error = 0;
2683     }

```

```

2684
2685     /* deal with port status changes */
2686     for (i = 1; i <= hub->descriptor->bNbrPorts; i++) {
2687         if (test_bit(i, hub->busy_bits))
2688             continue;
2689         connect_change = test_bit(i, hub->change_bits);
2690         if (!test_and_clear_bit(i, hub->event_bits) &&
2691             !connect_change && !hub->activating)
2692             continue;
2693
2694         ret = hub_port_status(hub, i,
2695                               &portstatus, &portchange);
2696         if (ret < 0)
2697             continue;
2698
2699         if (hub->activating && !hdev->children[i-1] &&
2700             (portstatus &
2701              USB_PORT_STAT_CONNECTION))
2702             connect_change = 1;
2703
2704         if (portchange & USB_PORT_STAT_C_CONNECTION) {
2705             clear_port_feature(hdev, i,
2706                               USB_PORT_FEAT_C_CONNECTION);
2707             connect_change = 1;
2708         }
2709
2710         if (portchange & USB_PORT_STAT_C_ENABLE) {
2711             if (!connect_change)
2712                 dev_dbg (hub_dev,
2713                         "port %d enable change, "
2714                         "status %08x\n",
2715                         i, portstatus);
2716             clear_port_feature(hdev, i,
2717                               USB_PORT_FEAT_C_ENABLE);
2718
2719             /*
2720             * EM interference sometimes causes badly
2721             * shielded USB devices to be shutdown by
2722             * the hub, this hack enables them again.
2723             * Works at least with mouse driver.
2724             */
2725             if (!(portstatus & USB_PORT_STAT_ENABLE)
2726                 && !connect_change
2727                 && hdev->children[i-1]) {
2728                 dev_err (hub_dev,
2729                         "port %i "
2730                         "disabled by hub (EMI?), "
2731                         "re-enabling...\n",
2732                         i);
2733                 connect_change = 1;
2734             }
2735         }
2736
2737         if (portchange & USB_PORT_STAT_C_SUSPEND) {
2738             clear_port_feature(hdev, i,
2739                               USB_PORT_FEAT_C_SUSPEND);
2740             if (hdev->children[i-1]) {
2741                 ret = remote_wakeup(hdev->
2742                                     children[i-1]);

```

```

2743         if (ret < 0)
2744             connect_change = 1;
2745     } else {
2746         ret = -ENODEV;
2747         hub_port_disable(hub, i, 1);
2748     }
2749     dev_dbg (hub_dev,
2750             "resume on port %d, status %d\n",
2751             i, ret);
2752 }
2753
2754 if (portchange & USB_PORT_STAT_C_OVERCURRENT) {
2755     dev_err (hub_dev,
2756             "over-current change on port %d\n",
2757             i);
2758     clear_port_feature(hdev, i,
2759                       USB_PORT_FEAT_C_OVER_CURRENT);
2760     hub_power_on(hub);
2761 }
2762
2763 if (portchange & USB_PORT_STAT_C_RESET) {
2764     dev_dbg (hub_dev,
2765             "reset change on port %d\n",
2766             i);
2767     clear_port_feature(hdev, i,
2768                       USB_PORT_FEAT_C_RESET);
2769 }
2770
2771 if (connect_change)
2772     hub_port_connect_change(hub, i,
2773                             portstatus, portchange);
2774 } /* end for i */
2775
2776 /* deal with hub status changes */
2777 if (test_and_clear_bit(0, hub->event_bits) == 0)
2778     ; /* do nothing */
2779 else if (hub_hub_status(hub, &hubstatus, &hubchange) < 0)
2780     dev_err (hub_dev, "get_hub_status failed\n");
2781 else {
2782     if (hubchange & HUB_CHANGE_LOCAL_POWER) {
2783         dev_dbg (hub_dev, "power change\n");
2784         clear_hub_feature(hdev, C_HUB_LOCAL_POWER);
2785         if (hubstatus & HUB_STATUS_LOCAL_POWER)
2786             /* FIXME: Is this always true? */
2787             hub->limited_power = 0;
2788         else
2789             hub->limited_power = 1;
2790     }
2791     if (hubchange & HUB_CHANGE_OVERCURRENT) {
2792         dev_dbg (hub_dev, "overcurrent change\n");
2793         msleep(500); /* Cool down */
2794         clear_hub_feature(hdev, C_HUB_OVER_CURRENT);
2795         hub_power_on(hub);
2796     }
2797 }
2798
2799 hub->activating = 0;
2800
2801 /* If this is a root hub, tell the HCD it's okay to

```



```

2802      * re-enable port-change interrupts now. */
2803      if (!hdev->parent && !hub->busy_bits[0])
2804          usb_enable_root_hub_irq(hdev->bus);
2805
2806 loop_autopm:
2807     /* Allow autosuspend if we're not going to run again */
2808     if (list_empty(&hub->event_list))
2809         usb_autopm_enable(intf);
2810 loop:
2811     usb_unlock_device(hdev);
2812     usb_put_intf(intf);
2813
2814 } /* end while (1) */
2815 }

```

## 14. 树，是什么样的树

USB 设备树是怎样一棵树？让我慢慢地道来。

hub\_events()里面第 2645 行，locktree()，用的就是汉诺塔里的那个经典思想——递归。

locktree()定义于 drivers/usb/core/hub.c:

```

990 static int locktree(struct usb_device *udev)
991 {
992     int t;
993     struct usb_device *hdev;
994
995     if (!udev)
996         return -ENODEV;
997
998     /* root hub is always the first lock in the series */
999     hdev = udev->parent;
1000     if (!hdev) {
1001         usb_lock_device(udev);
1002         return 0;
1003     }
1004
1005     /* on the path from root to us, lock everything from
1006      * top down, dropping parent locks when not needed
1007      */
1008     t = locktree(hdev);
1009     if (t < 0)
1010         return t;
1011
1012     /* everything is fail-fast once disconnect
1013      * processing starts
1014      */
1015     if (udev->state == USB_STATE_NOTATTACHED) {
1016         usb_unlock_device(hdev);
1017         return -ENODEV;
1018     }
1019 }

```

```

1020    /* when everyone grabs locks top->bottom,
1021    * non-overlapping work may be concurrent
1022    */
1023    usb_lock_device(udev);
1024    usb_unlock_device(hdev);
1025    return udev->portnum;
1026 }

```

传递进来的是这个 Hub 对应的 struct usb\_device 指针，995 行自然不必说。

999 行，parent，struct usb\_device 结构体的 parent 自然也是一个 struct usb\_device 指针。1000 行，判断 udev 的 parent 指针，你一定觉得奇怪，好像之前从来没有看到过 parent 指针，为何它突然之间出现了？它指向什么呀？Hub 驱动作为一个驱动程序，它并非是孤立存在的，没有主机控制器的驱动，没有 USB Core，Hub 驱动的存在将没有任何意义。其实我在前面就说过，Hub 准确地说应该是 Root Hub，它和主机控制器是绑定在一起的，专业一点说叫做“集成”在一起的。

因为 Hub 驱动不孤立，所以具体来说，作为 Root Hub，它的 parent 指针在主机控制器的驱动程序中就已经赋了值，这个值就是 NULL。换句话说，对于 Root Hub，它不需要再有父指针了，这个父指针本来就是给从 Root Hub 连出来的节点用的，这里这个函数名字叫做 locktree，顾名思义，锁住一棵树。这棵树就是 USB 设备树。很显然，USB 设备是从 Root Hub 开始，一个一个往外连的，比如 Root Hub 有 4 个端口，每个端口连一个 USB 设备，比如其中有一个是 Hub，那么这个 Hub 有可以继续有多个端口，于是一级一级地往下连，最终肯定会连成一棵树。

自从 Intel 提出了 EHCI 的规范以来，当今 USB 世界的发展趋势是：硬件厂商们总是让 EHCI 主机控制器里面拥有尽可能多的端口，换言之，就是希望大家别再用外接的 Hub 了，有一个 Root Hub 就够用了，也就是说，真的到了那种情况，USB 设备树的就不太像树了，顶多就是两级，一级是 Root Hub，下一级就是普通设备。严格来说，对于我们普通人来说，这样子也就够用了，假设你的 Root Hub 有 8 个端口，你说你够用不够用？鼠标，键盘，音响，U 盘，存储卡，8 个端口对普通人来说肯定够了。

所以说写代码也不是一件容易的事情，除了保证你的代码能让普通人正常使用，还得保证在其他情况下也能使用，locktree() 的想法就是这样。在 hub\_events() 里面加入 locktree() 的理由很简单，如果你的计算机里有两个 Hub，一个叫 parent，一个叫 child，child 接在 parent 的某个口上，那么 parent 在执行下面这段代码时，child 就不要去执行这段代码，否则会引起混乱。

为何会引起混乱？要知道，对于一个 Hub 来说，其所有正常的工作都是在 hub\_events() 这个函数中进行的，比如这些工作一种情况是删除一个子设备，这将有可能会直接导致 USB 设备树的拓扑结构发生变化，或者另一种情况，遍历整个子树去执行一个 resume 或者 reset 之类的操作，那么很显然，在这种情况下，一个 parent Hub 在进行这些操作时，不希望受到 child Hub 的影响。所以在这样一个政治背景下，2004 年的夏天，作为 Linux 内核开发中 USB 子系统的

三剑客之一的 David Brownell，决定加入 `locktree` 这个函数，这个函数的思想很简单，实际上就是借用了我国古代军事思想中的“擒贼先擒王”，用 David Brownell 本人的话说就是“lock parent first”。

每一个 Hub 在执行 `hub_events()` 中下面的那些代码时（特指 `locktree` 那个括号以下的那些代码），都得获得一把锁，锁住自己，而在锁住自己之前，又先得获得父亲的锁，确切地说，是尝试获得父亲的锁，如果能够获得父亲的锁，那么说明父亲当前没有执行 `hub_events()`（否则就没有办法获得父亲的锁），那么这种情况下子 Hub 才可以执行自己的 `hub_events()`。但是需要注意，在执行自己的代码之前，先把父 Hub 的锁给释放掉。因为我们说了，我们的目的是尝试获得父亲的锁，这个尝试的目的是为了保证在我们执行 `hub_events()` 之前的那一时刻，Parent Hub 并不是正在执行 `hub_events()`，而至于我们已经开始执行了 `hub_events()`，我们就不在乎 Parent Hub 是否也想开始执行 `hub_events()` 了。

Root Hub 就是整棵树的根，Root Hub 就没有 Parent Hub，所以，整个递归到了 Root Hub 就可以终止了。这也正是为什么 1000 行那句 if 语句，在判断出该设备是一个 Root Hub 之后，马上就执行锁住该设备。而如果不是 Root Hub，那么继续往下走，递归调用 `locktree()`，对于 `locktree()`，正常情况下它的返回值大于等于 0，所以小于 0 就算出错了。

然后 1015 行判断一下，如果我们把 Parent Hub 锁住了，可是自己却被断开了，即 `disconnect` 函数被执行了，那么就立刻停止，把 Parent Hub 的锁释放，然后返回把错误代码-`ENODEV`。

最后 1023 行，锁住自己，1024 行，释放父设备。

1025 行，返回当前设备的 `portnum`。`portnum` 就是端口号，你一定奇怪，没见过什么时候为 `portnum` 赋值了啊？别忘了这里是在讨论 Root Hub，对于 Root Hub 来说，它本身没有 `portnum` 这么一个概念，因为它不插在别的 Hub 的任何一个口上。所以对于 Root Hub 来说，它的 `portnum` 在主机控制器的驱动程序里给设置成了 0。而对于普通的 Hub，它的 `portnum` 在哪里赋的值呢？我们在后面就会看到的，别急。不过提醒一下，对于 Root Hub 来说，这里根本就不会执行到 1025 行来，刚才说了，对于 Root Hub，实际上在 1002 行那里就返回了，而且返回值就是 0。

就这样，我们看完了 `locktree()` 这个函数，接下来我们又该返回到 `hub_events()` 里面去了。

最终 `locktree()` 被加入到了内核中面，而且不止加了一处，除了加入到了 `hub_events()` 之外，在另外两个函数 `usb_suspend_device()` 和 `usb_resume_device()` 里面也有调用 `locktree()`。有趣的是，从 2.6.9 的内核一直到 2.6.22 的内核，我们都能看到 `locktree()` 这么一个函数，但是后来，`usb_suspend_device/usb_resume_device` 中没有了这个函数，就比如我们现在看到的 2.6.22 内核，搜索整个内核，只有 `hub_events()` 这一个地方调用了 `locktree()`，对此，Alan Stern 给出的说法是，内核中关于 USB 挂起的支持有了新的改进，不需要再调用 `locktree` 了。但是从 2.6.23 的内核开始，估计整个内核中就不会再有 `locktree` 了，Alan Stern 在这个夏天，提交了一个 patch，最终把 `locktree` 相关的代码全部从内核中移除了。

## 15. 没完没了的判断

2645 行就返回了，正常情况都是返回 0 或者正数，如果小于 0 那就说明失败了，在使用 interface 之前会调用 `usb_get_intf()` 来增加引用计数，而与之对应的是 `usb_put_intf()`，这里我们就调用了 `usb_put_intf()` 来减少引用计数。`continue` 的意思是开始新一轮 `while` 循环，如果 `hub_event_list` 里还有东西的话就继续处理。

2649 行，`usb_get_intfdata()`，判断一下，得到的是不是 hub，你问为什么得到的是 hub？回去看 `hub_probe()`，别忘了那时候我们调用过 `usb_set_intfdata` 从而把 `intf` 和 `hub` 联系起来了。那为什么还要判断？因为在 `hub_disconnect()` 中，有这么一句，`usb_set_intfdata(intf, NULL)`，而 `hub_events()` 和 `hub_disconnect()` 是异步执行的，就是说你执行你的，我执行我的，换言之，当 `hub_events()` 正执行时，`hub_disconnect()` 那边可能就已经取消了 `intf` 和 `hub` 之间建立起来的那层关系，所以这里需要判断一下。

2653 行，现在是时候该说一说 `USB_STATE_NOTATTACHED` 这个宏了，在 `include/linux/usb/ch9.h` 中：

```
557 enum usb_device_state {
558     /* NOTATTACHED isn't in the USB spec, and this state acts
559      * the same as ATTACHED ... but it's clearer this way.
560      */
561     USB_STATE_NOTATTACHED = 0,
562
563     /* chapter 9 and authentication (wireless) device states */
564     USB_STATE_ATTACHED,
565     USB_STATE_POWERED,                /* wired */
566     USB_STATE_UNAUTHENTICATED,        /* auth */
567     USB_STATE_RECONNECTING,          /* auth */
568     USB_STATE_DEFAULT,                /* limited function */
569     USB_STATE_ADDRESS,
570     USB_STATE_CONFIGURED,             /* most functions */
571
572     USB_STATE_SUSPENDED
573
574     /* NOTE: there are actually four different SUSPENDED
575      * states, returning to POWERED, DEFAULT, ADDRESS, or
576      * CONFIGURED respectively when SOF tokens flow again.
577      */
578 };
```

定义了一堆的宏，其中 `USB_STATE_NOTATTACHED` 的意思很明显，设备没有插在端口上。在代码里，有几个函数会把设备的状态设置成一个是汇报主机控制器异常死机的函数 `usb_hc_died()`，一个是 Hub 驱动自己提供的函数 `hub_port_disable()`，用于关掉一个端口的函数，或者用来断开设备的函数 `usb_disconnect()`，总之这几个函数只要它们执行了，那么设备肯定就没办法工作了，所以这里先判断设备的状态是不是 `USB_STATE_NOTATTACHED`，如果是那么就设置错误代码为 `-ENODEV`，然后调用 `hub_pre_reset()`，这个函数是与 `reset` 相关的。

2660 行, `usb_autopm_get_interface()`, 这个函数是 USB Core 提供的, 又是一个电源管理的函数, 这个函数所做的事情就是让 USB 接口的电源引用计数加一, 也就是说, 只要这个引用计数大于 0, 这个设备就不允许 `autosuspend`。

`autosuspend` 就是当用户在指定的时间内没有活动的话, 就自动挂起。应该说, 在 USB 中引入 `autosuspend/autoresume` 这还是最近的事情了, 最初有这个想法是在 2006 年的 5 月底, Alan Stern 在开源社区提出: 最近打算开始在 USB 里面实现对 `autosuspend/autoresume` 的支持。

所谓的 `autosuspend/autoresume`, 实际上是一种运行时的电源管理方式。而这些事情将由驱动程序来负责, 即当驱动程序觉得它的设备闲置了, 它就会触发 `suspend` 事件, 而当驱动程序要使用一个设备了, 但该设备正处于 `suspended` 状态, 那么驱动程序就会触发一个 `resume` 事件, `suspend` 事件和 `resume` 事件很显然是相对应的, 一个是挂起, 一个是恢复。

这里有一个很关键的理念, 又涉及了前面讲的那个设备树, 即, 当一个设备挂起时, 它必须通知它的 `parent`, 而 `parent` 就会决定看 `parent` 是不是也自动挂起, 反过来, 如果一个设备需要 “`resume`”, 那么它必须要求它的 `parent` 也 “`resume`”, 就是说这里有这么一种逻辑关系, 一个 `parent` 要想 “`suspend`”, 只有在它的 `children` 都 “`suspend`” 它才可以 “`suspend`”, 而一个 `child` 想要 “`resume`”, 只有在它的 `parent` 先 “`resume`” 了它才可以 “`resume`”。

还不明白? 举一个例子, 我和我的室友放寒假在复旦大学南区澡堂洗澡, 每人一个水龙头, 考虑到寒假期间洗澡的人数比较少, 管理员决定把水龙头的总闸调小, 但是也不能让我们正在洗的人洗不了, 所以只有满足了我们正在洗澡的人的水量, 才可以关小总闸。同样, 开学了以后, 大家都来洗澡, 可是总闸还是那么小, 那不行, 管理员就得调整总闸, 每个人调整自己的开关, 那样肯定没用, 总的流量就那么小, 所以这种情况下就得先开了总闸这样单个的开关的调节才有意义。

对于 Hub 来说, 当 `hub_events()` 处于运行的状态, 那么这个 Hub 接口就是在使用, 在这种情况下是不可以进行 “`autosuspend`” 的。对于这个上下文来说, `usb_autopm_get_interface()` 返回的就是 0。但是如果咱们的 Hub 是处于 `suspended` 状态, 那么这里首先就会把 Hub 唤醒, 即会执行 `resume`。先不多说了, 继续往下看吧。

2667 行, 判断 `quiescing`, 以前咱们说过, `struct usb_hub` 里面有两个成员, `quiescing` 和 `activating`, 并且在 `hub_activate()` 中已经看到了, 我们把 `quiescing` 设置成了 0, 而把 `activating` 设置成了 1。现在是时候来说一说这两个变量的含义了。我们说了 `quiescing` 是停止的意思, 在 `reset` 时我们会设置它为 1, 在 `suspend` 时我们也会把它设置为 1, 一旦把它设置成了 1, 那么 hub 驱动程序就不会再提交任何 `urb`, 而如果我们把 `activating`, 那么 Hub 驱动程序就会给每个端口发送一个叫做 `Get Port Status` 的请求, 通常情况下, Hub 驱动只有在一个端口发生了状态变化的情况下才会去发送 `Get Port Status` 从而去获得端口的状态。所以就是说, 正常情况下, 这两个 `flag` 都是不会设置的。即正常情况下这两个 `flag` 都应该是 0。

2671 行, 在这个情景下, `hub->error` 当然是 0, 但是如果今后我们正式工作以后, 再次来到这里的话, `hub->error` 可能就不再是 0 了。对于那种情况, 需要调用 `usb_reset_composite_device()`, 这个函数是咱们自己定义的, 目的就是把设备 reset, 下面来具体看一下, 来自 `drivers/usb/core/hub.c`:

```

3062 int usb_reset_composite_device(struct usb_device *udev,
3063                               struct usb_interface *iface)
3064 {
3065     int ret;
3066     struct usb_host_config *config = udev->actconfig;
3067
3068     if (udev->state == USB_STATE_NOTATTACHED ||
3069         udev->state == USB_STATE_SUSPENDED) {
3070         dev_dbg(&udev->dev, "device reset not allowed in state %d\n",
3071               udev->state);
3072         return -EINVAL;
3073     }
3074
3075     /* Prevent autosuspend during the reset */
3076     usb_autoresume_device(udev);
3077
3078     if (iface && iface->condition != USB_INTERFACE_BINDING)
3079         iface = NULL;
3080
3081     if (config) {
3082         int i;
3083         struct usb_interface *cintf;
3084         struct usb_driver *drv;
3085
3086         for (i = 0; i < config->desc.bNumInterfaces; ++i) {
3087             cintf = config->interface[i];
3088             if (cintf != iface)
3089                 down(&cintf->dev.sem);
3090             if (device_is_registered(&cintf->dev) &&
3091                 cintf->dev.driver) {
3092                 drv = to_usb_driver(cintf->dev.driver);
3093                 if (drv->pre_reset)
3094                     (drv->pre_reset)(cintf);
3095             }
3096         }
3097     }
3098
3099     ret = usb_reset_device(udev);
3100
3101     if (config) {
3102         int i;
3103         struct usb_interface *cintf;
3104         struct usb_driver *drv;
3105
3106         for (i = config->desc.bNumInterfaces - 1; i >= 0; --i) {
3107             cintf = config->interface[i];
3108             if (device_is_registered(&cintf->dev) &&
3109                 cintf->dev.driver) {
3110                 drv = to_usb_driver(cintf->dev.driver);
3111                 if (drv->post_reset)
3112                     (drv->post_reset)(cintf);

```

```

3113         }
3114         if (cintf != iface)
3115             up(&cintf->dev.sem);
3116     }
3117 }
3118
3119 usb_autosuspend_device(udev);
3120 return ret;
3121 }

```

usb\_autoresume\_device()增加设备的引用计数，禁止设备 autosuspend 的发生。

后面的 usb\_autosuspend\_device()则刚好相反，减少设备的引用计数，并且使得设备可以被 autosuspend。

3068 行，在设备处于 USB\_STATE\_NOTATTACHED 或者 USB\_STATE\_SUSPENDED 的状态时，reset 是不被允许的。

3078 行，别忘了我们传递给 usb\_reset\_composite\_device 的有两个参数，一个是设备，一个是接口。而 USB\_INTERFACE\_BINDING 是一个宏，来自 include/linux/usb.h:

```

83 enum usb_interface_condition {
84     USB_INTERFACE_UNBOUND = 0,
85     USB_INTERFACE_BINDING,
86     USB_INTERFACE_BOUND,
87     USB_INTERFACE_UNBINDING,
88 };

```

这是表示接口的状态的，BINDING 就表示正在和驱动绑定。最开始，在 USB Core 发现初始化设备时，但是在 hub\_probe 被调用之前，接口是处于 USB\_INTERFACE\_BINDING 状态的，直到 hub\_probe 结束了之后，接口则是处于 USB\_INTERFACE\_BOUND 状态，即所谓的绑定好了，而如果 hub\_probe 出错了，那么接口就将处于 USB\_INTERFACE\_UNBOUND 状态。

我们这里 config 是 struct usb\_host\_config 结构体指针，被赋为 udev->actconfig，对于一个设备来说，它使用的是什么配置，这个在初始化时就设置好了的。

struct usb\_host\_config 结构体中有一个结构体 struct usb\_config\_descriptor desc，表示配置描述符，还有一个 struct usb\_interface \*interface[USB\_MAXINTERFACES] 数组，USB\_MAXINTERFACES 定义为 32。所以 config->desc.bNumInterfaces 及 config->interface[] 数组的意思就很明确了。3086 行的这个 for 循环就是说，这个设备有几个接口就一个一个遍历，cintf 作为临时变量来表示每一个 struct usb\_interface。首先我们要明白，usb\_reset\_composite\_device()这个函数我们可是既指定了设备又指定了接口的，那么 3088 行判断 cintf 不等于 iface 是什么意思呢？

回到刚才那个 3078 行，如果 iface 不处在 BINDING 的情况下，我们将 iface 设置为 NULL 了，而这里 cintf 是从 config->interface[] 数组里得出来的值，它肯定不为 NULL，那么这里如果

`cintf` 不等于 `iface`，就说明 `iface` 之前是不处于 `BINDING` 的状态，对于这种情况我们需要执行 3089 行，这样做的原因是为了等待。`dev` 是 `struct device` 结构体，是 `struct usb_interface` 结构体的一个成员，而 `sem` 是 `struct device` 结构体的一个信号量，`struct semaphore sem`，这个信号量专门用于同步，之所以这里需要信号量，原因如下：既然我们现在针对的是一个设备多个接口的情况，那么势必就有这样一种可能：一个设备多个接口，每个接口对应一个驱动，那么当设备 `fail` 时，有可能每个驱动都希望能够 `reset`，对于这种情况，我们当然需要保证这个过程不要出现混乱，于是设置一个信号量就好了，你 `reset` 时我就不能 `reset`，我 `reset` 时你就不能 `reset`。

那么为什么要单独把 `USB_INTERFACE_BINDING` 列出来呢？注释里说得很清楚了，当一个接口的状态处于 `BINDING` 时，其实就是这个接口对应的驱动的 `probe()` 函数正在执行，这个时候实际上已经获得锁了。`probe` 函数是提供给 USB Core 调用的，在 USB Core 中，调用 `probe()` 的前后也有这么一对 `down()/up()` 函数。

因为 `probe()` 这个操作也忌讳被别人影响，所以说这里对于正处于 `BINDING` 状态的接口就不需要再获得锁了，或者说不需要获得信号量了。否则就将是一个经典的死锁问题，我第一次遇到内核 Bug 就是一次死锁问题，在 8250 串口驱动中，明明已经有锁了，还要再次去获取锁，结果系统就死机了。

另一个问题需要清楚的是，`usb_reset_composite_device()` 这个函数的诞生是因为目前越来越多的设备都是复合设备，即一个设备中有多个接口。内核中引入这个函数是在 2006 年夏天，Alan Stern 又一次向社区里提出一个新的理念，即原本对于每个设备来说，都可以调用函数 `usb_reset_device` 来执行 `reset` 操作，而当今发展趋势是让一个设备包含多个接口，而我们知道一个驱动对应一个接口，于是就出现了多个驱动对应一个设备的情况，那么一个 `usb_reset_device()` 函数就有可能对所有的接口都造成影响，于是，Alan 利用这样两个函数——`struct usb_driver` 结构体中两个成员函数 `pre_reset` 和 `post_reset`，我们知道每个 `struct usb_driver` 都有两个成员 `pre_reset` 和 `post_reset`，而 Alan 的理念是每个驱动定义自己的 `pre_reset` 和 `post_reset`，当我们在调用 `usb_reset_device` 之前，先遍历调用每个驱动的 `pre_reset`，Alan 称这些个 `pre_reset` 给每个绑定在该设备上的驱动一个警告，告诉它们，要“reset”了。

在执行完 `usb_reset_device` 之后，再遍历调用每个驱动的 `post_reset`，`post_reset`，其作用是让每个驱动知道，`reset` 完成了。另一方面，`post_reset` 还有一个作用，因为 `reset` 会把设备原来的状态都给清除掉，所以 `post_reset` 就担负了这么一个使命，即重新初始化设备。但是你得明白，并不是每一个设备驱动都定义了 `pre_reset` 和 `post_reset`，有没有必要执行这两个操作那都是自己决定，你要是无所谓，觉得“reset”整个设备对你这个接口没什么影响，不为这两个指针赋值也可以。这也就是为什么在 3093 行和 3111 行这两处要判断这两个指针是否被赋了值，只有赋了值才去执行相应的函数，否则就没有必要。

继续看，`device_is_registered()` 是一个内联函数，就是判断 `struct device` 结构体指针的一个成员 `is_registered` 是否为 1，这个值对于 Root Hub 来说，在主机控制器驱动程序中初始化时把



它设置为 1，对于普通的设备来说，以后我们会看到，在 Hub 驱动为其作初始化时也会设置为 1。而 `dev.driver` 也是在初始化时会赋值，特别对于 Hub，这个 `dev.driver` 就是与之对应的 `struct device_driver` 结构体。而 3092 行这个 `to_usb_driver()` 是一个宏，它得到的就是与之对应的 `struct usb_driver` 结构体，而对于 Hub 来说，这就是 `struct usb_driver hub_driver`，所以，我们就不难知道，3094 行及后面 3112 行所做的就是调用 `hub_driver` 里面的两个成员函数——`hub_pre_reset()` 和 `hub_post_reset()`。

总之，3094 行，3099 行，3012 行，这三个函数的调用，就是真正地完成了一次 Hub 的 reset，就相当于重启一次计算机，重启的原因是我们遇见了 `hub->error`。这三个函数的细节我们先暂时不看，以后再看。需要强调的一点是，3101 行到 3117 行这一段代码，和 3081 行到 3097 行这一段代码，基本上是对称的。

3120 行，`return ret` 返回了。返回值就是 `usb_reset_device` 的返回值。

回到 `hub_events()` 之后，立刻把 `hub->nerrors` 和 `hub->error` 给复位了，设置为 0。其中 `nerrors` 是记录发生错误的次数，`nerrors` 就是 `number of errors`，要是连续发生错误就每次让 `nerrors` 加 1。

从下面开始就进入 Hub 驱动中的代码了。可以说在 `hub_events()` 中，此前的每一行代码都显得非常的枯燥，让人根本看不明白 Hub 驱动究竟是干什么的，直到下面这些代码才真正诠释了一个 Hub 驱动应该做的事情。我们需要明白，Hub 的存在不是为了它自己，我们不是为了用 Hub 而买 Hub，我们是为了让 Hub 连接真正想用的设备。

---

## 16. 一个都不能少

2686 行，终于发现从这里开始针对端口进行分析了，有几个端口就对几个端口进行分析，分析每一个端口的状态变化，一个都不能少。很显然，这就是我们期待看到的代码，我们马上就可以知道，当我们把一个 USB 设备插入 USB 端口后 USB 设备提供的那些接口函数究竟是如何被调用的，特别是 `probe` 函数。

`bNbrports` 是前面我们获得的 Hub 描述符的一个成员，表示这个 Hub 有几个端口。很显然，USB 设备却不可以没有描述符。这里就是遍历每一个端口。`busy_bits` 是 `struct usb_hub` 的一个成员，`unsigned long busy_bits[1]`，接下来的 `event_bits` 也是，`change_bits` 也是 USB-hub，`unsigned long event_bits[1]`，`unsigned long change_bits[1]`，`test_bit()` 我们太熟悉了。这里我们要测试的有三个设置，首先测试 `busy_bits`，这个 flag 实际上只有在 `reset` 和 `resume` 的函数内部才会设置，而这里的意思是，如果眼下这个端口正在执行 `reset` 或者 `resume` 操作，那么咱们就跳过去，不予理睬。

2689 行，测试 `change_bits`。结合 2690 行，2691 行，2692 行一起看。如果这个端口对应的 `change_bits` 没有设置，`event_bits` 没有设置过，`hub->activating` 也为 0，那么这里就执行 `continue`，不过我们想都不用想，因为我们就是从 `hub_activate` 进来的。我们来时 `activating` 就是设置成了 1 的，所以这里的 `continue` 是不用执行的。换言之，我们继续往下走。

2694 行，`hub_port_status()`，`portstatus` 和 `portchange` 是我们在 `hub_events()` 伊始定义的两个变量，`u16 portstatus`，`u16 portchange` 都是 16 位的。尽管说了很多遍了，但是我还是得再说第一遍，这个函数仍然是来自 `drivers/usb/core/hub.c`：

```
1413 static int hub_port_status(struct usb_hub *hub, int port1,
1414                             u16 *status, u16 *change)
1415 {
1416     int ret;
1417
1418     mutex_lock(&hub->status_mutex);
1419     ret = get_port_status(hub->hdev, port1, &hub->status->port);
1420     if (ret < 4) {
1421         dev_err (hub->intfdev,
1422                 "%s failed (err = %d)\n", __FUNCTION__, ret);
1423         if (ret >= 0)
1424             ret = -EIO;
1425     } else {
1426         *status = le16_to_cpu(hub->status->port.wPortStatus);
1427         *change = le16_to_cpu(hub->status->port.wPortChange);
1428         ret = 0;
1429     }
1430     mutex_unlock(&hub->status_mutex);
1431     return ret;
1432 }
```

重要的是其中的 `get_port_status()` 函数：

```
300 /*
301  * USB 2.0 spec Section 11.24.2.7
302  */
303 static int get_port_status(struct usb_device *hdev, int port1,
304                             struct usb_port_status *data)
305 {
306     int i, status = -ETIMEDOUT;
307
308     for (i = 0; i < USB_STS_RETRIES && status == -ETIMEDOUT; i++) {
309         status = usb_control_msg(hdev, usb_rcvctrlpipe(hdev, 0),
310                                 USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_PORT, 0, port1,
311                                 data, sizeof(*data), USB_STS_TIMEOUT);
312     }
313     return status;
314 }
```

现在我们再也不会对 `usb_control_msg()` 函数陌生了，这个函数做什么的我们完全是一目了然。Get Port Status 是 Hub 的一个标准请求，对我们来说就是一次控制传输就可以完成。这个请求的格式如图 2.16.1 所示。

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100011B	GET_STATUS	0	端口号	4	端口状态 和变化状态

图 2.16.1 Get Port Status 请求

其中这个 GET\_STATUS 的对应具体的数值可以在 spec 的 Table 11-16 中对比得到。而关于各种请求，在 include/linux/usb/ch9.h 中也定义了相应的宏，

```
79 #define USB_REQ_GET_STATUS 0x00
80 #define USB_REQ_CLEAR_FEATURE 0x01
81 #define USB_REQ_SET_FEATURE 0x03
82 #define USB_REQ_SET_ADDRESS 0x05
83 #define USB_REQ_GET_DESCRIPTOR 0x06
84 #define USB_REQ_SET_DESCRIPTOR 0x07
85 #define USB_REQ_GET_CONFIGURATION 0x08
86 #define USB_REQ_SET_CONFIGURATION 0x09
87 #define USB_REQ_GET_INTERFACE 0x0A
88 #define USB_REQ_SET_INTERFACE 0x0B
89 #define USB_REQ_SYNCH_FRAME 0x0C
90
91 #define USB_REQ_SET_ENCRYPTION 0x0D /* Wireless USB */
92 #define USB_REQ_GET_ENCRYPTION 0x0E
93 #define USB_REQ_RPIPE_ABORT 0x0E
94 #define USB_REQ_SET_HANDSHAKE 0x0F
95 #define USB_REQ_RPIPE_RESET 0x0F
96 #define USB_REQ_GET_HANDSHAKE 0x10
97 #define USB_REQ_SET_CONNECTION 0x11
98 #define USB_REQ_SET_SECURITY_DATA 0x12
99 #define USB_REQ_GET_SECURITY_DATA 0x13
100 #define USB_REQ_SET_WUSB_DATA 0x14
101 #define USB_REQ_LOOPBACK_DATA_WRITE 0x15
102 #define USB_REQ_LOOPBACK_DATA_READ 0x16
103 #define USB_REQ_SET_INTERFACE_DS 0x17
```

比如这里传递给 usb\_control\_msg 的请求就是 USB\_REQ\_GET\_STATUS，它的值为 0，和 spec 中定义的 GET\_STATUS 的值是对应的。这个请求返回两个值，一个称为 Port Status,一个称为 Port Change Status。usb\_control\_msg()的调用注定了返回值将保存在 struct usb\_port\_status \*data 里面，这个结构体定义于 drivers/usb/core/hub.h 中：

```
58 /*
59  * Hub Status and Hub Change results
60  * See USB 2.0 spec Table 11-19 and Table 11-20
61  */
62 struct usb_port_status {
63     __le16 wPortStatus;
64     __le16 wPortChange;
65 } __attribute__((packed));
```

很显然这个格式是和实际的 spec 对应的，我们给 get\_port\_status() 传递的实参是

&hub->status->port, port 也是一个 struct usb\_port\_status 结构体变量, 所以在 hub\_port\_status() 里面, 1426 行和 1427 行我们就得到了 Status Bits 和 Status Change Bits。get\_port\_status()返回值就是 Get Port Status 请求的返回数据的长度, 它至少应该能够保存 wPortStatus 和 wPortChange, 所以至少不能小于 4, 所以 1420 行有这个错误判断。这样, hub\_port\_status()就返回了, 而 status 和 change 这两个指针也算是满载而归了, 正常的话返回值就是 0。

继续往下走, 2699 行, children[i-1], 我们从没有见过它, 但是我想大家都知道, 正是像 parent 和 children 这样的指针才能把 USB 树给建立起来, 而我们才刚上路, 肯定还没有设置 children, 所以对我们来说, 至少目前 children 数组肯定为空, 而我们又知道 hub->activating 这时候肯定为 1, 所以就看第三个条件了, portstatus&USB\_PORT\_STAT\_CONNECTION, 这是什么意思? 这表明这个端口连接了设备, 没错, USB\_PORT\_STAT\_CONNECTION 这个宏定义于 drivers/usb/core/hub.h 中:

```

67 /*
68  * wPortStatus bit field
69  * See USB 2.0 spec Table 11-21
70  */
71 #define USB_PORT_STAT_CONNECTION      0x0001
72 #define USB_PORT_STAT_ENABLE          0x0002
73 #define USB_PORT_STAT_SUSPEND         0x0004
74 #define USB_PORT_STAT_OVERCURRENT     0x0008
75 #define USB_PORT_STAT_RESET           0x0010
76 /* bits 5 to 7 are reserved */
77 #define USB_PORT_STAT_POWER           0x0100
78 #define USB_PORT_STAT_LOW_SPEED       0x0200
79 #define USB_PORT_STAT_HIGH_SPEED      0x0400
80 #define USB_PORT_STAT_TEST            0x0800
81 #define USB_PORT_STAT_INDICATOR       0x1000
82 /* bits 13 to 15 are reserved */
83
84 /*
85  * wPortChange bit field
86  * See USB 2.0 spec Table 11-22
87  * Bits 0 to 4 shown, bits 5 to 15 are reserved
88  */
89 #define USB_PORT_STAT_C_CONNECTION    0x0001
90 #define USB_PORT_STAT_C_ENABLE        0x0002
91 #define USB_PORT_STAT_C_SUSPEND       0x0004
92 #define USB_PORT_STAT_C_OVERCURRENT   0x0008
93 #define USB_PORT_STAT_C_RESET         0x0010

```

这都是这两个变量对应的宏, spec 里面对这些宏的意义说得很清楚。USB\_PORT\_STAT\_CONNECTION 的意思的确是表示是否有设备连接在这个端口上, 我们不妨假设有, 那么 portstatus 和它相与的结果就是 1, 在 spec 里面, 我们会看到 connect\_change 被设置成了 1。

而接下来, USB\_PORT\_STAT\_C\_CONNECTION 则是表示这个端口的 Current Connect Status 位是否有变化, 如果有变化, 那么 portchange 和 USB\_PORT\_STAT\_C\_CONNECTION 相

与的结果就是 1，对于这种情况，我们需要发送另一个请求以清除这个 flag，并且将 connect\_change 也设置为 1。这个请求叫做 Clear Port Feature。这个请求也是 Hub 的标准请求，它的作用就是 reset hub 端口的某种 feature。clear\_port\_feature()定义于 drivers/usb/core/hub.c:

```
162 /*
163  * USB 2.0 spec Section 11.24.2.2
164  */
165 static int clear_port_feature(struct usb_device *hdev, int port1, int
feature)
166 {
167     return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
168         USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port1,
169         NULL, 0, 1000);
170 }
```

USB\_REQ\_CLEAR\_FEATURE 和 spec 中的 CLEAR\_FEATURE 请求是对应的，那么一共有哪些 feature 呢？在 drivers/usb/core/hub.h 中是这样定义的。

```
38 /*
39  * Port feature numbers
40  * See USB 2.0 spec Table 11-17
41  */
42 #define USB_PORT_FEAT_CONNECTION      0
43 #define USB_PORT_FEAT_ENABLE          1
44 #define USB_PORT_FEAT_SUSPEND         2
45 #define USB_PORT_FEAT_OVER_CURRENT    3
46 #define USB_PORT_FEAT_RESET           4
47 #define USB_PORT_FEAT_POWER           8
48 #define USB_PORT_FEAT_LOWSPEED        9
49 #define USB_PORT_FEAT_HIGHSPEED       10
50 #define USB_PORT_FEAT_C_CONNECTION    16
51 #define USB_PORT_FEAT_C_ENABLE        17
52 #define USB_PORT_FEAT_C_SUSPEND       18
53 #define USB_PORT_FEAT_C_OVER_CURRENT  19
54 #define USB_PORT_FEAT_C_RESET         20
55 #define USB_PORT_FEAT_TEST            21
56 #define USB_PORT_FEAT_INDICATOR       22
```

而在 spec 中，Table 11-17 与之相对应定义了许多的 feature，我们清除的正是 C\_PORT\_CONNECTION 这一个 feature。

spec 里面说了，清除一个状态改变的 feature 就等于承认这么一个 feature。(clearing that status change acknowledges the change) 理由很简单，每次你检测到一个 flag 被设置之后，你都应该清除掉它，以便下次别人设置你就知道是有人设置了，否则你不知道在你下次判断是不是又有人设置了。同理，接下来的每个与 portchange 相关的判断语句都要这么做。所以如果 portchange 与上 USB\_PORT\_STAT\_C\_CONNECTION 确实为 1，那么我们就需要清除这个 feature。同时我们当然也要记录 connect\_change 为 1。

继续，每个端口都有一个开关，这叫做 enable 或者 disable 一个端口。portchange 和 USB\_PORT\_STAT\_C\_ENABLE 相与结果如果为 1 的话，说明端口开关有变化。和刚才一样，

首先我们要做的是，清除掉这个变化的 feature。但是这里需要注意，spec 里对这个 feature 是这样规定的，如果 `portchange` 和 `USB_PORT_STAT_C_ENABLE` 为 1，说明这个端口是从 `enable` 状态进入了 `disable` 状态。

为什么呢？因为在 spec 规定了，Hub 的端口是不可以直接设置成 `enable` 的。通常让 Hub 端口设置成 `enable` 的方法是“reset hub port”，用 spec 的话说，这叫做发送另一个请求，名为 `SET_FEATURE`。`SET_FEATURE` 和 `CLEAR_FEATURE` 是对应的，一个用于设置一个用于清除。对于 `PORT_ENABLE`，用 spec 里的话说：“This bit may be set only as a result of a `SetPortFeature(PORT_RESET)` request.” `PORT_RESET` 是为 Hub 定义的众多 feature 中的一种。

最后提醒一点，2711 行至 2715 行这段 if 语句仅仅是为了打印调试信息的，也就是说如果端口 `enable` 改变了，但是端口连接没有改变，那么就打印信息来通知调试者，不要把 `clear_port_feature` 这一行也纳入到 if 语句里去了。因为端口 `enable` 的改变有多种可能，其中一种可能就是由于检测到了 `disconnection`，这种情况，就是我们下面要处理的。

下面这段代码就比较复杂了，电磁干扰都给扯进来了。EMI 也就是电磁干扰。就是说有时 Hub 端口的 `enable` 变成 `disable` 有可能是由于电磁干扰造成的。这个 if 条件判断的是：端口被 `disable` 了，但是连接没有变化，并且 `hdev->children[i]` 还有值，这就说明明明有子设备连接在端口上，可是端口却被 `disable` 了，基本上这种情况就是电磁干扰造成的，否则 Hub 端口不会有这种举动。那么这种情况就设置 `connect_change` 为 1。因为接下来我们会看到对于 `connect_change` 为 1 的情况，会专门进行处理，而直白的说法就是，`hub_events()` 其实最重要的任务就是对于端口连接有变化的情况进行处理，或者说进行响应。

再往下，`portchange` 和 `USB_PORT_STAT_C_SUSPEND` 相与结果如果为 1，表明连在该端口的设备的 `suspend` 状态有变化，并且是从 `suspended` 状态出来，也就是说 `resume` 完成。（别问我为什么，spec 就这么规定的，没什么理由）那么首先我们就调用 `clear_port_feature` 清掉这个 feature。总之这里做的就是对于该端口连了子设备的情况就把子设备唤醒，否则如果端口没有连接子设备，那么就把端口 `disable` 掉。

2754 行，`portchange` 如果和 `USB_PORT_STAT_C_OVERCURRENT` 相与结果为 1 的话，说明这个端口可能曾经存在电流过大的情况，而现在这种情况不存在了，或者本来不存在而现在存在了。对此我们能做的就是首先清除这个 feature。有一种比较特别的情况，如果其他端口电流过大，那么将会导致本端口断电，即 Hub 上一个端口出现 `over-current` 条件将有可能引起 Hub 上其他端口陷入 `powered off` 的状态。不管怎么说，对于 `over-current` 的情况我们都把 Hub 重新上电，执行 `hub_power_on()`。

2763 行，`portchange` 如果和 `USB_PORT_STAT_C_RESET` 相与为结果 1 的话，这叫做一个端口从 `Resetting` 状态进入到 `Enabled` 状态。

2771 行，`connect_change` 如果为 1，就执行 `hub_port_connect_change()`，这是每一个看 Hub

驱动的人最期待的函数，因为这正是我们的原始动机，即当一个 USB 设备插入 USB 接口之后究竟会发生什么，USB 设备驱动程序提供的 probe 函数究竟是如何被调用的。这些疑问统统会在这个函数中得到答案。这个函数来自 drivers/usb/core/hub.c:

```

2412 static void hub_port_connect_change(struct usb_hub *hub, int port1,
2413                                     u16 portstatus, u16 portchange)
2414 {
2415     struct usb_device *hdev = hub->hdev;
2416     struct device *hub_dev = hub->intfdev;
2417     u16 wHubCharacteristics =
2418         le16_to_cpu(hub->descriptor->wHubCharacteristics);
2419     int status, i;
2420     dev_dbg (hub_dev,
2421             "port %d, status %04x, change %04x, %s\n",
2422             port1, portstatus, portchange, portspeed (portstatus));
2423
2424     if (hub->has_indicators) {
2425         set_port_led(hub, port1, HUB_LED_AUTO);
2426         hub->indicator[port1-1] = INDICATOR_AUTO;
2427     }
2428
2429     /* Disconnect any existing devices under this port */
2430     if (hdev->children[port1-1])
2431         usb_disconnect(&hdev->children[port1-1]);
2432     clear_bit(port1, hub->change_bits);
2433
2434 #ifdef CONFIG_USB_OTG
2435     /* during HNP, don't repeat the debounce */
2436     if (hdev->bus->is_b_host)
2437         portchange &= ~USB_PORT_STAT_C_CONNECTION;
2438 #endif
2439
2440     if (portchange & USB_PORT_STAT_C_CONNECTION) {
2441         status = hub_port_debounce(hub, port1);
2442         if (status < 0) {
2443             if (printk_ratelimit())
2444                 dev_err (hub_dev, "connect-debounce failed, "
2445                         "port %d disabled\n", port1);
2446             goto done;
2447         }
2448         portstatus = status;
2449     }
2450
2451     /* Return now if nothing is connected */
2452     if (!(portstatus & USB_PORT_STAT_CONNECTION)) {
2453
2454         /* maybe switch power back on (e.g. root hub was reset) */
2455         if ((wHubCharacteristics & HUB_CHAR_LPSM) < 2
2456             && !(portstatus & (1 << USB_PORT_FEAT_POWER)))
2457             set_port_feature(hdev, port1, USB_PORT_FEAT_POWER);
2458
2459         if (portstatus & USB_PORT_STAT_ENABLE)
2460             goto done;
2461         return;
2462     }
2463

```

```

2464 #ifdef CONFIG_USB_SUSPEND
2465     /* If something is connected, but the port is suspended, wake it up.*/
2466     if (portstatus & USB_PORT_STAT_SUSPEND) {
2467         status = hub_port_resume(hub, port1, NULL);
2468         if (status < 0) {
2469             dev_dbg(hub_dev,
2470                 "can't clear suspend on port %d; %d\n",
2471                 port1, status);
2472             goto done;
2473         }
2474     }
2475 #endif
2476
2477     for (i = 0; i < SET_CONFIG_TRIES; i++) {
2478         struct usb_device *udev;
2479
2480         /* reallocate for each attempt, since references
2481          * to the previous one can escape in various ways
2482          */
2483         udev = usb_alloc_dev(hdev, hdev->bus, port1);
2484         if (!udev) {
2485             dev_err(hub_dev,
2486                 "couldn't allocate port %d usb_device\n",
2487                 port1);
2488             goto done;
2489         }
2490
2491         usb_set_device_state(udev, USB_STATE_POWERED);
2492         udev->speed = USB_SPEED_UNKNOWN;
2493         udev->bus_mA = hub->mA_per_port;
2494         udev->level = hdev->level + 1;
2495
2496         /* set the address */
2497         choose_address(udev);
2498         if (udev->devnum <= 0) {
2499             status = -ENOTCONN;    /* Don't retry */
2500             goto loop;
2501         }
2502
2503         /* reset and get descriptor */
2504         status = hub_port_init(hub, udev, port1, i);
2505         if (status < 0)
2506             goto loop;
2507
2508         /* consecutive bus-powered hubs aren't reliable; they can
2509          * violate the voltage drop budget.  if the new child has
2510          * a "powered" LED, users should notice we didn't enable it
2511          * (without reading syslog), even without per-port LEDs
2512          * on the parent.
2513          */
2514         if (udev->descriptor.bDeviceClass == USB_CLASS_HUB
2515             && udev->bus_mA <= 100) {
2516             u16    devstat;
2517
2518             status = usb_get_status(udev, USB_RECIP_DEVICE, 0,
2519                                     &devstat);
2520             if (status < 2) {
2521                 dev_dbg(&udev->dev, "get status %d ?\n", status);
2522                 goto loop_disable;

```



```

2523     }
2524     le16_to_cpus(&devstat);
2525     if ((devstat & (1 << USB_DEVICE_SELF_POWERED)) == 0) {
2526         dev_err(&udev->dev,
2527             "can't connect bus-powered hub "
2528             "to this port\n");
2529         if (hub->has_indicators) {
2530             hub->indicator[port1-1] =
2531                 INDICATOR_AMBER_BLINK;
2532             schedule_delayed_work (&hub->leds, 0);
2533         }
2534         status = -ENOTCONN;    /* Don't retry */
2535         goto loop_disable;
2536     }
2537 }
2538
2539 /* check for devices running slower than they could */
2540 if (le16_to_cpu(udev->descriptor.bcdUSB) >= 0x0200
2541     && udev->speed == USB_SPEED_FULL
2542     && highspeed_hubs != 0)
2543     check_highspeed (hub, udev, port1);
2544
2545 /* Store the parent's children[] pointer. At this point
2546  * udev becomes globally accessible, although presumably
2547  * no one will look at it until hdev is unlocked.
2548  */
2549 status = 0;
2550
2551 /* We mustn't add new devices if the parent hub has
2552  * been disconnected; we would race with the
2553  * recursively_mark_NOTATTACHED() routine.
2554  */
2555 spin_lock_irq(&device_state_lock);
2556 if (hdev->state == USB_STATE_NOTATTACHED)
2557     status = -ENOTCONN;
2558 else
2559     hdev->children[port1-1] = udev;
2560 spin_unlock_irq(&device_state_lock);
2561
2562 /* Run it through the hoops (find a driver, etc) */
2563 if (!status) {
2564     status = usb_new_device(udev);
2565     if (status) {
2566         spin_lock_irq(&device_state_lock);
2567         hdev->children[port1-1] = NULL;
2568         spin_unlock_irq(&device_state_lock);
2569     }
2570 }
2571
2572 if (status)
2573     goto loop_disable;
2574
2575 status = hub_power_remaining(hub);
2576 if (status)
2577     dev_dbg(hub_dev, "%d mA power budget left\n", status);
2578
2579 return;
2580
2581 loop_disable:

```

```

2582     hub_port_disable(hub, port1, 1);
2583 loop:
2584     ep0_reinit(udev);
2585     release_address(udev);
2586     usb_put_dev(udev);
2587     if (status == -ENOTCONN)
2588         break;
2589 }
2590
2591 done:
2592     hub_port_disable(hub, port1, 1);
2593 }

```

我打算不像前面那样一行一行讲了，我必须先来一个提纲挈领，开门见山把这个函数的思想讲清楚，否则一行一行往下讲肯定会晕头转向的。

## 17. 盖茨家对 Linux 代码的影响

`hub_port_connect_change`，顾名思义，当 Hub 端口上有连接变化时调用这个函数，这种变化既可以是物理变化也可以是逻辑变化。注释里说得也很清楚。有三种情况会调用这个函数，第一种情况是连接有变化，第二种情况是端口本身重新使能，即所谓的 `enable`，这种情况通常就是为了对付电磁干扰的，正如我们前面的判断中所说的那样，第三种情况就是在复位一个设备时发现其描述符变了，这通常对应的是硬件本身有了升级。很显然，第一种情况是真正的物理变化，后两者就算是逻辑变化。

前面几行的赋值不用多说，2422 行的 `portspeed()` 函数就是获得这个端口所接的设备的 `speed`，来自 `drivers/usb/core/hub.c`：

```

121 static inline char *portspeed(int portstatus)
122 {
123     if (portstatus & (1 << USB_PORT_FEAT_HIGHSPEED))
124         return "480 Mb/s";
125     else if (portstatus & (1 << USB_PORT_FEAT_LOWSPEED))
126         return "1.5 Mb/s";
127     else
128         return "12 Mb/s";
129 }

```

这个函数意图太明显了，就是确定是高速、低速，还是全速的设备。这些信息都包含在 `portstatus` 的 `bit9` 和 `bit10` 里面。

2424 行这一小段，关于指示灯的设置，有指示灯就设置一下，指示灯可以设置成琥珀色，可以设置成绿色，可以设置成关闭，也可以设置成自动，这里设置为自动。所谓自动设置就是 Hub 自己根据端口的状态来设置，比如 Hub 挂起了，那么指示灯当然就应该熄灭。

2430 行，如果这个 Hub 的子设备还有值，那么什么也别说了，先把它给切断了。

我们说了，端口连接有变化，可是变化可以是两个方向的。一个是原来没有设备现在有了，另一个是恰恰相反，原来有设备而现在没有。对于前者，`hdev->children[port1-1]`肯定为空，而对于后者这个指针应该就不为空；对于前者，我们接下来要做的事情就是对新连接进来的设备进行初始化配置，分配地址然后为该设备寻找相应的设备驱动程序，如果找到合适的驱动程序了就调用该设备驱动程序提供的指针函数来再进行更细致更深入更对口的初始化。但是对于后者，就没必要那么麻烦了，直接调用 `usb_disconnect()` 函数执行一些清扫工作，并且把 Hub 的 `change_bits` 清除掉。然后再确定一下端口上确实没有连接什么设备，就可以返回了。

2434 行到 2438 行这一段我们不管，因为我们早已经作了一个假设，假设我们关闭了 `CONFIG_USB_OTG` 这个编译开关。

2440 行，别忘了，对于连接从有到无的变化，刚才我们可是清掉了 Hub 的 `change_bits`，所以这里再次判断 `portchange&USB_PORT_STAT_C_CONNECTION` 的意思就是对于连接从无到有有这种情况，我们还必须判断反弹。

什么是反弹？换一种说法叫做去抖动。比如你在网上聊 QQ，你按一下键，正常情况下，在按下键盘和松开键盘的过程中触片可能快速地接触和分离好多次，而此时此刻你自己有些激动或紧张，那么按键肯定是多次抖动，而驱动程序不可能响应你很多次，因为你毕竟只是按一下键盘。同理，这样的去抖动技术在 Hub 中也是需要的。所以原则上，spec 规定，只有持续了 100 ms 的插入才算真正的插入，或者说才算稳定的插入。`hub_port_debounce` 就是干这件事情的，这个函数会让你至少等待 100 ms，如果设备依然在，那么说明稳定了，这种情况下函数返回值就是端口的状态，即 2448 行中把 `status` 赋给 `portstatus`。而如果 100 ms 不到设备又被拔走了，那么返回负的错误码，然后打印信息告诉你发生错误。

`printk_ratelimit` 是一个新的函数，其实就是 `printk` 的变种，`printk_ratelimit` 的用途就是当某条消息可能会重复地被多次打印的，甚至打印成千上万条，直接导致日志文件溢出，把别的信息都冲掉了，所以这样是不好的情况，于是进化出来一个 `printk_ratelimit()`，它会控制打印消息的频率，如果短期内连续出现打印消息，那么它把消息抛弃，这种情况下这个函数返回 0，所以，只有返回非 0 值的情况下才会真正打印信息。

2452 行，如果经历过以上操作之后，现在端口状态已经是没有设备连接了，那么再做两个判断，一个判断如果是该端口的电源被关闭了，那么就打开电源。另一个判断如果是该端口还处于 `ENABLE` 的状态，那么 `goto done`。可以看到 `done` 那里的代码就是把这个端口给 `disable` 掉。否则，如果端口已经是 `disable` 状态了，那么就直接返回吧，这次事件就算处理完了。

2466 行，这一段就是说连接虽然变化了，并且设备也是从无到有了，但是如果之前这个端口是出于 `suspend` 状态的话，那么这里就要调用 `hub_port_resume()` 把这个端口给恢复。关于电源管理的深层次代码，我们暂时先搁一边。但是作为 Hub 驱动程序，电源管理部分是必不可少

的。

2477 行，SET\_CONFIG\_TRIES 这个宏，这又要引出一段故事了。

以下这个 for 循环的目的很明确，为一个 USB 设备申请内存空间，设置它的状态，把它复位，为它设置地址，获取它的描述符，然后向设备模型中添加这个设备，以后会为这个设备寻找它的驱动程序，驱动程序提供的 probe() 函数就会被调用。

但是对于 USB 来说，只要调用 device\_add 这么一个函数向设备模型核心层添加设备就够了，剩下的事情设备模型层会去处理，这就是设备模型的优点，所以也叫统一的设备模型。就是说不管是 PCI、USB 还是 SCSI，总线驱动的工作就是申请并建立总线的数据结构，而设备驱动的工作就是往这条总线上注册，调用 driver\_add，而设备这边也是一样，也往该总线上注册，即调用 device\_add。而 driver\_add 就会在总线上寻找每一个设备，如果找到了自己支持的设备，并且该设备没有和别的驱动相绑定，那么就绑定该设备。反过来，设备这边的做法也一样，device\_add 在总线上寻找每一个设备驱动，找到了合适的就绑定该设备。最后，调用 probe 函数，将“兵权”就交给了设备驱动。整个这个过程就叫做 USB 设备初始化。

所以，对于设备驱动程序本身来讲，它只是参与了设备初始化的一部分，很小的一部分，真正重要的工作，都是由核心来执行，对于 USB 设备来说，就是 Hub 驱动来集中处理这些事情。之所以这些工作可以统一来做，是因为凡是 USB 设备，它都必须遵循一些共同的特征。

那么这个过程为何要循环呢？以前 Linux 内核中是没有 SET\_CONFIG\_TRIES 这个宏的，后来在 2003 年的圣诞节前夕，David Brownell 提交了一个内核补丁，出现了这个宏，并且把这个宏的值被固定为 2。2004 年底，这个宏的定义发生了一些变化，变得更加灵活。

第一，为什么将这个宏的值设为 2？这个理由很简单，我们说了，要获取设备描述符。如何获得？给设备发送一个 GET-DEVICE-DESCRIPTOR 的请求，然后设备就会返回设备描述符。

再回到 USB，本来 spec 就规定了只要发送这个请求设备就该返回设备描述符，可是这么一试，它却说请求失败。所以，发这些重要的请求都按两次发，一次不成功再发一次，成功了就不发，两次还不成功，那没办法了。

第二，后来为何这个宏的值又改变了？我们来看，现在这个宏被改是为了什么？在 drivers/usb/core/hub.c 中：

```
1446 #define PORT_RESET_TRIES      5
1447 #define SET_ADDRESS_TRIES      2
1448 #define GET_DESCRIPTOR_TRIES    2
1449 #define SET_CONFIG_TRIES        (2 * (use_both_schemes + 1))
1450 #define USE_NEW_SCHEME(i)       ((i) / 2 == old_scheme_first)
```

看 usb\_both\_schemes 和 old\_scheme\_first 这两个参数：

```
109 static int old_scheme_first = 0;
```

```
110 module_param(old_scheme_first, bool, S_IRUGO | S_IWUSR);
111 MODULE_PARM_DESC(old_scheme_first,
112     "start with the old device initialization scheme");
113
114 static int use_both_schemes = 1;
115 module_param(use_both_schemes, bool, S_IRUGO | S_IWUSR);
116 MODULE_PARM_DESC(use_both_schemes,
117     "try the other device initialization scheme if the "
118     "first one fails");
```

这两个参数是模块加载时的参数，在你用 `modprobe` 或者 `insmod` 加载一个模块时，你可以指定 `old_scheme_first` 的值和 `usb_both_schemes` 的值，如果你不指定，那么这里的默认值是 `old_scheme_first` 为 0，而 `use_both_schemes` 为 1。

那么它们具体起着什么作用？`scheme`，方案，计划的意思。那么 `old scheme`，`both scheme` 这两个词组似乎已经反映出来有两个方案，一个是旧的，一个是新的。什么方面的方案？一个是来自 Linux 的方案，一个是来自 Windows 的方案，目的是为了获得设备的描述符。

关于方案问题，这里需要一些背景知识。

第一个，端点 0。0 号端点是 `spec` 中一个特殊的端点。`spec` 中是这样规定的，所有的 USB 设备都有一个默认的控制管道。英文叫 `Default Control Pipe`，与这条管道对应的端点叫做 0 号端点，也就是传说中的 `endpoint zero`。这个端点的信息不需要记录在配置描述符里，也就是说并没有一个专门的端点描述符来描述这个 0 号端点，原因是端点 0 基本上所有的特性都是在 `spec` 规定好了的，大家都一样，所以不需要每个设备另外准备一个描述符来描述它。（换言之，在接口描述符里的 `bNumEndpoints` 是指该接口包含的端点，但是这其中并不包含端点 0，如果一个设备除了这个端点 0 以外没有别的端点了，那么它的接口描述符里的 `bNumEndpoints` 就应该是 0，而不是 1。）

然而，别忘了我说的是“基本上”，有一个特性则是不一样的，这叫做 `maximum packet size`，每个端点都有这么一个特性，即告诉你该端点能够发送或者接收的包的最大值。对于通常的端点来说，这个值被保存在该端点描述符中的 `wMaxPacketSize` 这一个 `field`，而对于端点 0 就不一样了，由于它自己没有一个描述符，而每个设备又都有这么一个端点，所以这个信息被保存在了设备描述符里，所以我们在设备描述符里可以看到 `bMaxPacketSize0`，而且 `spec` 还规定了，这个值只能是 8，16，32 或者 64 这四者之一，而且，如果一个设备工作在高速模式，这个值还只能是 64，取别的值都不行。

而我们知道，我们所做的很多工作都是通过与端点 0 打交道来完成的，那么端点 0 的 `max packet size` 自然是我们必须要知道的，否则肯定没法正确地进行控制传输。于是问题就出现了。我不知道 `max packet size` 就没法进行正常的传输，可是 `max packet size` 又在设备描述符里，不进行传输我就不知道 `max packet size` 啊？

我刚才说了，`max packet size` 只能是 8，16，32 或者 64，这也就是说，至少你得是 8，而

我们惊奇地发现，设备描述符公共是 18 个字节，其中，第 Byte7 恰恰就是 bMaxPacketSize0，Byte0 到 Byte7 算起来就是 8 个字节，那也就是说，我先读 8 个字节，然后设备返回 device descriptor 的前 8 个字节，于是我就知道它真正的 max packet size 了，于是我再读一次，这次才把整个描述符 18 个字节都给读出来，不就行了吗？

在 2004 年 10 月，开源社区的同志们发现一个怪事，Sony 的一个摄像机没法在 Linux 下正常工作。问题就出在获取设备描述符上，当你把一个 8 个字节的 GET-DEVICE-DESCRIPTOR 的请求发送给 sony 的设备时，你不能得到一个 8 个字节的完整的描述符，相反，你会遇到溢出的错误，因为 Sony 的设备只想一口气把 18 个字节的整个设备描述符全都给返回，结果导致了错误，而且实践证明这样的错误还有可能会毁坏设备。

这怎么办？有人说这款设备在 Windows 下工作是完全正常的。Windows 采取的是另一种策略，或者说方案，就是直接发送 64 个字节的请求过去，即要求你的设备返回 64 个字节过来，如果你设备端点 0 的 max packet size 是 32 或者 64，那么你反正只要把 18 个字节的设备描述传递过来就可以了，但是如果你的设备端点 0 的 max packet size 是 8 或者 16，而设备描述符是 18 个字节，一次肯定传递不完，那么你必然是传递了一次以后还等待着继续传递。但是从驱动角度来说，我只要获得了 8 个字节就够了，而对于设备，你不是等着继续传吗，我直接对你做一次 reset，让你复位，这样不就清掉了你剩下想传的数据了吗？然后获得了前 8 个字节就可以知道你真正的 max packet size，再就按这个真正的最大值来进行下面的传输，首先就是获得那个 18 个字节的真正完整的设备描述符。这样子，也就达到了目的了。这就是 Windows 下面的处理方法。

于是开源社区的兄弟们发现，很多厂商都是按着 Windows 的这种策略来测试自己的设备的，他们压根儿就没有测试过请求 8 个字节的设备描述符，于是，也就没人能保证当你发送请求要它返回 8 个字节的设备描述符时它能够正确地响应。所以，Linux 开发人员们委曲求全，把这种 Windows 下的策略给加了进来，其实，Linux 的那种策略才是 USB spec 提供的策略，而现实是，Windows 没有遵守这种策略，然而厂商们出厂时就只是测试了能在 Windows 下工作，他们认为遵守 Windows 就是遵守了 spec。而事实却并非如此。严格意义来说，这是 Windows 这边的 Bug，不过这种做法却引导了潮流。

所以，就这样，如今的代码里实现了这两种策略，每种试两次，原来的那种策略叫做 old scheme，现在的做法就是具体使用那种策略作为用户你可以在加载模块时自己设置，但是如果不设置，那么默认的方法就是先使用新的这种机制，试两次，然后如果不行，就使用旧的那种，我们看到前面讲到的宏 USE\_NEW\_SCHEME，它就是用来判断是不是使用新的 scheme 的。这个宏会在 hub\_port\_init() 函数中用到，如果它为真，那么就用新的策略，即发送期望 64 个字节的请求。如果 fail 了，发送那个 8 个字节的请求。

至此我们总算可以理解 SET\_CONFIG\_TRIES 这个宏了，它被定义为  $(2 * (\text{use\_both\_schemes} + 1))$ ，而 use\_both\_schemes 就是说两种策略都用，这个参数也是可以自己在加载模块时设置，

默认值为 1，即默认的情况时先用新的策略，不行就用旧的。而 `usb_both_schemes` 为 1 就意味着 `SET_CONFIG_TRIES` 等于 4，即旧的策略试两次，新的策略试两次，当然，成功了就不用多试了，多试是为失败而准备的。

看明白了这个宏，我们可以进入到这个 `for` 循环来仔细看个究竟了。

---

## 18. 八大重量级函数闪亮登场（一）

其实 Hub 在 USB 世界里扮演的又何尝不是这种角色呢？我们来看这个 `for` 循环，这是 Hub 驱动中最核心的代码，然而这段代码却自始至终是在为别的设备服务。

在这个循环中，主要涉及八个重量级函数，先点明它们的角色分工。

第一个函数，`usb_alloc_dev()`，一个 `struct usb_device` 结构体指针，申请内存，这个结构体指针可不是为 Hub 准备的，它正是为了 Hub 这个端口所接的设备而申请的，别忘了我们此时此刻的上下文，之所以进入到了这个循环，是因为我们的 Hub 检测到某个端口有设备连接了进来，所以我们作为 Hub 驱动当然就义不容辞的要为该设备做点什么。

第二个函数，`usb_set_device_state()`，这个函数用来设置设备的状态，在 `struct usb_device` 结构体中，有一个成员 `enum usb_device_state state`，这一刻，会把这个设备的状态设置为 `USB_STATE_POWERED`，即上电状态。

第三个函数，`choose_address()`，为设备选择一个地址。一会咱们会用实例来查看效果。

第四个函数，`hub_port_init()`，端口初始化，主要就是前面所讲的获取设备的描述符。

第五个函数，`usb_get_status()`，这个函数是专门为 Hub 准备的，不是为当前的这个 Hub，而是说当前 Hub 的这个端口上连接的结果又是 Hub，那么和连接普通设备当然不一样。

第六个函数，`check_highspeed()`，不同速度的设备当然待遇不一样。

第七个函数，`usb_new_device()`。寻找驱动程序，调用驱动程序的 `probe`，跟踪这个函数就能一直到设备驱动程序的 `probe()` 函数的调用。

第八个函数，`hub_power_remaining()`，电源管理。

下面就让我们来逐个看一看这八个函数。不过因为 `usb_alloc_dev()` 函数作为设备生命线的开端已经在 USB Core 部分中讲过了，所以这里从第二个函数 `usb_set_device_state()` 开始。

`usb_set_device_state()` 在 `drivers/usb/core/hub.c` 文件中定义，鉴于 `drivers/usb/core/hub.c` “出

镜频率”过高，从此以后在 Hub 部分里，凡是出自 drivers/usb/core/hub.c 这个文件的函数将不再做介绍其来源，这个就当是默认的位置。

```

1062 void usb_set_device_state(struct usb_device *udev,
1063                          enum usb_device_state new_state)
1064 {
1065     unsigned long flags;
1066
1067     spin_lock_irqsave(&device_state_lock, flags);
1068     if (udev->state == USB_STATE_NOTATTACHED)
1069         ; /* do nothing */
1070     else if (new_state != USB_STATE_NOTATTACHED) {
1071
1072         /* root hub wakeup capabilities are managed out-of-band
1073          * and may involve silicon errata ... ignore them here.
1074          */
1075         if (udev->parent) {
1076             if (udev->state == USB_STATE_SUSPENDED
1077                 || new_state == USB_STATE_SUSPENDED)
1078                 ; /* No change to wakeup settings */
1079             else if (new_state == USB_STATE_CONFIGURED)
1080                 device_init_wakeup(&udev->dev,
1081                                   (udev->actconfig->desc.bmAttributes
1082                                    & USB_CONFIG_ATT_WAKEUP));
1083             else
1084                 device_init_wakeup(&udev->dev, 0);
1085         }
1086         udev->state = new_state;
1087     } else
1088         recursively_mark_NOTATTACHED(udev);
1089     spin_unlock_irqrestore(&device_state_lock, flags);
1090 }

```

这个函数不是很长，问题是，这个函数中又调用了别的函数。

USB\_STATE\_NOTATTACHED，意思是设备已经断开了，这种情况当然什么也不用做。

因为在 usb\_alloc\_dev 设置了等于 USB\_STATE\_ATTACHED，所以继续往下看，new\_state，结合实参看一下，传递的是 USB\_STATE\_POWERED，Root Hub 另有管理方式，我们这里首先就处理非 Root Hub 的情况，如果原来就是 USB\_STATE\_SUSPENDED，现在还设置 USB\_STATE\_SUSPENDED，那么当然什么也不用做。如果新的状态要被设置为 USB\_STATE\_CONFIGURED，那么调用 device\_init\_wakeup()，初始化唤醒。

要认识 device\_init\_wakeup()首先需要知道两个概念，can\_wakeup 和 should\_wakeup。这两个家伙从哪里来的？看 struct device 结构体，里面有这么一个成员， struct dev\_pm\_info power，来看一看 struct dev\_pm\_info，来自 include/linux/pm.h 文件：

```

265 struct dev_pm_info {
266     pm_message_t          power_state;
267     unsigned               can_wakeup:1;
268 #ifdef CONFIG_PM
269     unsigned               should_wakeup:1;

```



```

270     pm_message_t      prev_state;
271     void              * saved_state;
272     struct device      * pm_parent;
273     struct list_head   entry;
274 #endif
275 };

```

这些都是电源管理部分的核心数据结构，显然我们没有必要深入研究，只是需要知道，`can_wakeup` 为 1 时表明一个设备可以被唤醒，设备驱动为了支持 Linux 中的电源管理，有责任调用 `device_init_wakeup()` 来初始化 `can_wakeup`。而 `should_wakeup` 则是在设备的电源状态发生变化时被 `device_may_wakeup()` 用来测试，测试它该不该变化。因此 `can_wakeup` 表明的是一种能力，`should_wakeup` 表明的是有了这种能力以后去不去做某件事。

我们给 `device_init_wakeup()` 传递的参数是这个设备，以及配置描述符中的 `bmAttributes&USB_CONFIG_ATT_WAKEUP`。这是因为，`spec` 中的 Table 9-10 规定了，`bmAttributes` 的 D5 表明的就是一个 USB 设备是否具有被唤醒的能力。而 `USB_CONFIG_ATT_WAKEUP` 被定义为 `1<<5`，所以这里比较的就是 D5 是否为 1，为 1 就是说：“我能”。

1083 行中的 `else` 的意思是，如果设备将要被设置的新状态又不是 `USB_STATE_CONFIGURED`，那么就执行这里的 `device_init_wakeup`，这里第二个参数传递的是 0，也就是说先不打开这个设备的 `wakeup` 能力。这个上下文就是这种情况，刚刚才说到，咱们的新状态就是 `USB_STATE_POWERED`。

直到 1086 行，才正式把状态设置为新的状态，对于这里的这个上下文，那就是 `USB_STATE_POWERED`。

1087 行这里又是一个 `else`，那么很显然这个 `else` 的意思就是原来的状态不是 `USB_STATE_NOTATTACHED` 而现在要设置成 `USB_STATE_NOTATTACHED`。这又是一个递归函数。其作用就是像其名字中所说的那样，递归的把各设备都设置成 `NOTATTACHED` 状态。

```

1028 static void recursively_mark_NOTATTACHED(struct usb_device *udev)
1029 {
1030     int i;
1031
1032     for (i = 0; i < udev->maxchild; ++i) {
1033         if (udev->children[i])
1034             recursively_mark_NOTATTACHED(udev->children[i]);
1035     }
1036     if (udev->state == USB_STATE_SUSPENDED)
1037         udev->discon_suspended = 1;
1038     udev->state = USB_STATE_NOTATTACHED;
1039 }

```

这段代码真的是太简单了，属于递归函数的经典例子。就是每一个设备遍历自己的子节点，一个个调用 `recursively_mark_NOTATTACHED()` 函数把其 `state` 设置为

USB\_STATE\_NOTATTACHED。如果设备的状态处于 USB\_STATE\_SUSPENDED，那么设置 udev->discon\_suspended 为 1，struct usb\_device 中的一个成员，unsigned discon\_suspended，含义很明显，表示 Disconnected while suspended。这里这么一设置，暂时我们还不知道有什么用，不过到时候我们就会在电源管理部分的代码里看到判断这个 flag 了，很显然设置了这个 flag 就会阻止 suspend 相关的代码被调用。

## 19. 八大重量级函数闪亮登场（二）

在开始第三个函数前，2492 行至 2494 行这三行代码对 udev 中的 speed，bus\_mA，level 进行赋值。

先说一下，bus\_mA，struct usb\_device 中的成员 unsigned short bus\_mA 记录的是能够从总线上获得的电流，毫无疑问就是咱们前面算出来的 hub 上的那个 mA\_per\_port。

level，级别，表示 Usb 设备树的级连关系。Root Hub 当然其 level 就是 0，其下面一层就是 level 1，再下面一层就是 level 2，依此类推。

然后说 speed，include/linux/usb/ch9.h 中定义了这么一个枚举类型的变量：

```
548 /* USB 2.0 defines three speeds, here's how Linux identifies them */
549
550 enum usb_device_speed {
551     USB_SPEED_UNKNOWN = 0,           /* enumerating */
552     USB_SPEED_LOW, USB_SPEED_FULL,   /* usb 1.1 */
553     USB_SPEED_HIGH,                  /* usb 2.0 */
554     USB_SPEED_VARIABLE,              /* wireless (usb 2.5) */
555 };
```

很明显的含义，用来标志设备的速度。众所周知，USB 设备有三种速度，低速、全速及高速。USB1.1 只有低速，全速，后来才出现了高速，高速就是所谓的 480 MB/s，不过细心的你或许注意到这里还有一个 USB\_SPEED\_VARIABLE。

2005 年 5 月，Intel 等公司推出了 Wireless USB spec 1.0 版，即所谓的无线 USB 技术，这个 USB 技术也称为 USB 2.5。无线技术的推出必然会让设备的速度不再稳定，当年这个标准推出时是号称在 3 米范围内，能够提供 480 MB/s 的理论传输速度，而在 10 米范围左右出现递减，据说是 10 米内变为 110 MB/s。那时正值英特尔在中国成立 20 周年，所以中国这边的员工每人发了一个无线 USB 鼠标。其实就是一个 USB 接头，接在计算机的 USB 端口上，而鼠标这边没有线，鼠标和接头之间的通信是无线的，使用蓝牙技术。总之，这里的变量 usb\_device\_speed 就是用来表示设备速度的，在现阶段还不知道这个设备究竟是什么速度的，所以先设置为 UNKNOWN。等到知道了以后再进行真正的设置。

第三个函数，choose\_address()。

这个函数的目的就是为设备选择一个地址。很显然，要通信就要有地址，你要给人写信，你首先得知道人家的通信地址，或者电子邮箱地址。

```

1132 static void choose_address(struct usb_device *udev)
1133 {
1134     int                devnum;
1135     struct usb_bus    *bus = udev->bus;
1136
1137     /* If khubd ever becomes multithreaded, this will need a lock */
1138
1139     /* Try to allocate the next devnum beginning at bus->devnum_next. */
1140     devnum = find_next_zero_bit(bus->devmap.devicemap, 128,
1141                                bus->devnum_next);
1142     if (devnum >= 128)
1143         devnum = find_next_zero_bit(bus->devmap.devicemap, 128, 1);
1144
1145     bus->devnum_next = ( devnum >= 127 ? 1 : devnum + 1);
1146
1147     if (devnum < 128) {
1148         set_bit(devnum, bus->devmap.devicemap);
1149         udev->devnum = devnum;
1150     }
1151 }

```

那么现在是时候让我们来认识一下 USB 子系统里面关于地址的游戏规则了。而在 USB 世界里，一条总线就是大树一棵，一个设备就是一片叶子。为了记录这棵树上的每一片叶子节点，每条总线设有一个地址映射表，即 struct usb\_bus 结构体里有一个成员 struct usb\_devmap devmap。

```

269 struct usb_devmap {
270     unsigned long devicemap[128 / (8*sizeof(unsigned long))];
271 };

```

同时 struct usb\_bus 结构体里面还有一个成员——int devnum\_next，在总线初始化时，其 devnum\_next 被设置为 1，而在 struct usb\_device 中有一个 int devnum，这个 choose\_address 函数的基本思想就是一个轮询的算法。

我们来介绍一下这段代码背后的思想。首先，bus 上面有这么一张表，假设 unsigned long=4Bytes，那么 unsigned long devicemap[128/(8\*sizeof(unsigned long))]就等价于 unsigned long devicemap[128/(8\*4)]，进而等价于 unsigned long devicemap[4]，而 4Bytes 就是 32 个 bits，因此这个数组最终表示的就是 128bits。而这也对应于一条总线可以连接 128 个 Usb 设备。之所以这里使用 sizeof(unsigned long)，就是为了跨平台应用，不管 unsigned long 到底是几，总之这个 devicemap 数组最终可以表示 128 位。

在 128 个 bits 里，每当加入一个设备，就先找到下一位为 0 的 bit，然后把该 bit 设置为 1，同时把 struct usb\_device 中的 devnum 设置为该数字，比如我们找到第 19 位为 0，那么就把 devnum

设置为 19，同时把 bit 19 设置为 1，而 struct usb\_bus 中的 devnum\_next 就设置为 20。

那么所谓轮询，即如果这个编号超过了 128 位，那么就从 1 开始继续搜索，因为也许开始那段的号码原来分配给某个设备但后来这个设备撤掉了，所以这个号码将被设置为 0，于是再次可用。

弄清楚了这些基本思想后，我们再来看代码就很简单了。

find\_next\_zero\_bit()的意思很明显，名字解释了一切。不同的体系结构提供了不同的函数实现，比如 i386，这个函数就定义于 arch/i386/lib/bitops.c 中，而 x86 64 位则对应于 arch/x86\_64/lib/bitops.c 中，利用这个函数我们就可以找到这 128 位中下一个为 0 的那一位。这个函数的第三个参数表示从哪里开始寻找，我们注意到第一次我们是从 devnum\_next 开始找，如果最终返回值“暴掉”了(大于或者等于 128)，那么就从 1 开始再找一次。而 bus->devnum\_next 也是按我们说的那样设置，正常就是 devnum+1，但如果 devnum 已经达到 127 了，那么就从头再来，设置为 1。

如果 devnum 正常，那么就把 bus 中的 device map 中的那一位设置为 1。同时把 udev->devnum 设置为 devnum。然后这个函数就可以返回了。如果 128 个 bits 都被占用了，devnum 就将是 0 或者负的错误码，于是 choose\_address 返回之后我们就要进行判断。

## 20. 八大重量级函数闪亮登场（三）

接下来我们来到了第四个函数，hub\_port\_init()，这个函数和接下来要遇到的 usb\_new\_device()是最重要的两个函数，也是相对复杂的函数。

```

2105 static int
2106 hub_port_init (struct usb_hub *hub, struct usb_device *udev, int port1,
2107                int retry_counter)
2108 {
2109     static DEFINE_MUTEX(usb_address0_mutex);
2110
2111     struct usb_device      *hdev = hub->hdev;
2112     int                    i, j, retval;
2113     unsigned                delay = HUB_SHORT_RESET_TIME;
2114     enum usb_device_speed  oldspeed = udev->speed;
2115     char                    *speed, *type;
2116
2117     /* root hub ports have a slightly longer reset period
2118      * (from USB 2.0 spec, section 7.1.7.5)
2119      */
2120     if (!hdev->parent) {
2121         delay = HUB_ROOT_RESET_TIME;
2122         if (port1 == hdev->bus->otg_port)
2123             hdev->bus->b_hnp_enable = 0;

```

```

2124     }
2125
2126     /* Some low speed devices have problems with the quick delay, so */
2127     /* be a bit pessimistic with those devices. RHbug #23670 */
2128     if (oldspeed == USB_SPEED_LOW)
2129         delay = HUB_LONG_RESET_TIME;
2130
2131     mutex_lock(&usb_address0_mutex);
2132
2133     /* Reset the device; full speed may morph to high speed */
2134     retval = hub_port_reset(hub, port1, udev, delay);
2135     if (retval < 0)          /* error or disconnect */
2136         goto fail;
2137     retval = -ENODEV;
2138
2139     if (oldspeed != USB_SPEED_UNKNOWN && oldspeed != udev->speed) {
2140         dev_dbg(&udev->dev, "device reset changed speed!\n");
2141         goto fail;
2142     }
2143     oldspeed = udev->speed;
2144
2145     /* USB 2.0 section 5.5.3 talks about ep0 maxpacket ...
2146      * it's fixed size except for full speed devices.
2147      * For Wireless USB devices, ep0 max packet is always 512 (tho
2148      * reported as 0xff in the device descriptor). WUSB1.0[4.8.1].
2149      */
2150     switch (udev->speed) {
2151     case USB_SPEED_VARIABLE:          /* fixed at 512 */
2152         udev->ep0.desc.wMaxPacketSize = __constant_cpu_to_le16(512);
2153         break;
2154     case USB_SPEED_HIGH:              /* fixed at 64 */
2155         udev->ep0.desc.wMaxPacketSize = __constant_cpu_to_le16(64);
2156         break;
2157     case USB_SPEED_FULL:              /* 8, 16, 32, or 64 */
2158         /* to determine the ep0 maxpacket size, try to read
2159          * the device descriptor to get bMaxPacketSize0 and
2160          * then correct our initial guess.
2161          */
2162         udev->ep0.desc.wMaxPacketSize = __constant_cpu_to_le16(64);
2163         break;
2164     case USB_SPEED_LOW:               /* fixed at 8 */
2165         udev->ep0.desc.wMaxPacketSize = __constant_cpu_to_le16(8);
2166         break;
2167     default:
2168         goto fail;
2169     }
2170
2171     type = "";
2172     switch (udev->speed) {
2173     case USB_SPEED_LOW:               speed = "low"; break;
2174     case USB_SPEED_FULL:              speed = "full"; break;
2175     case USB_SPEED_HIGH:              speed = "high"; break;
2176     case USB_SPEED_VARIABLE:          speed = "variable";
2177     speed = "variable";
2178     type = "Wireless ";
2179     break;
2180     default:                          speed = "?"; break;
2181     }
2182     dev_info (&udev->dev,

```

```

2184         "%s %s speed %sUSB device using %s and address %d\n",
2185         (udev->config) ? "reset" : "new", speed, type,
2186         udev->bus->controller->driver->name, udev->devnum);
2187
2188     /* Set up TT records, if needed */
2189     if (hdev->tt) {
2190         udev->tt = hdev->tt;
2191         udev->ttport = hdev->ttport;
2192     } else if (udev->speed != USB_SPEED_HIGH
2193                && hdev->speed == USB_SPEED_HIGH) {
2194         udev->tt = &hub->tt;
2195         udev->ttport = port1;
2196     }
2197
2198     /* Why interleave GET_DESCRIPTOR and SET_ADDRESS this way?
2199      * Because device hardware and firmware is sometimes buggy in
2200      * this area, and this is how Linux has done it for ages.
2201      * Change it cautiously.
2202      *
2203      * NOTE: If USE_NEW_SCHEME() is true we will start by issuing
2204      * a 64-Byte GET_DESCRIPTOR request. This is what Windows does,
2205      * so it may help with some non-standards-compliant devices.
2206      * Otherwise we start with SET_ADDRESS and then try to read the
2207      * first 8 Bytes of the device descriptor to get the ep0 maxpacket
2208      * value.
2209      */
2210     for (i = 0; i < GET_DESCRIPTOR_TRIES; (++i, msleep(100))) {
2211         if (USE_NEW_SCHEME(retry_counter)) {
2212             struct usb_device_descriptor *buf;
2213             int r = 0;
2214
2215             #define GET_DESCRIPTOR_BUFSIZE 64
2216             buf = kmalloc(GET_DESCRIPTOR_BUFSIZE, GFP_NOIO);
2217             if (!buf) {
2218                 retval = -ENOMEM;
2219                 continue;
2220             }
2221
2222             /* Retry on all errors; some devices are flakey.
2223              * 255 is for WUSB devices, we actually need to use
2224              * 512 (WUSB1.0[4.8.1]).
2225              */
2226             for (j = 0; j < 3; ++j) {
2227                 buf->bMaxPacketSize0 = 0;
2228                 r = usb_control_msg(udev, usb_rcvaddr0pipe(),
2229                                     USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
2230                                     USB_DT_DEVICE << 8, 0,
2231                                     buf, GET_DESCRIPTOR_BUFSIZE,
2232                                     USB_CTRL_GET_TIMEOUT);
2233                 switch (buf->bMaxPacketSize0) {
2234                     case 8: case 16: case 32: case 64: case 255:
2235                         if (buf->bDescriptorType ==
2236                             USB_DT_DEVICE) {
2237                             r = 0;
2238                             break;
2239                         }
2240                     /* FALL THROUGH */
2241                     default:
2242                         if (r == 0)

```

```

2243             r = -EPROTO;
2244             break;
2245         }
2246         if (r == 0)
2247             break;
2248     }
2249     udev->descriptor.bMaxPacketSize0 =
2250         buf->bMaxPacketSize0;
2251     kfree(buf);
2252
2253     retval = hub_port_reset(hub, port1, udev, delay);
2254     if (retval < 0)          /* error or disconnect */
2255         goto fail;
2256     if (oldspeed != udev->speed) {
2257         dev_dbg(&udev->dev,
2258             "device reset changed speed!\n");
2259         retval = -ENODEV;
2260         goto fail;
2261     }
2262     if (r) {
2263         dev_err(&udev->dev, "device descriptor "
2264             "read/%s, error %d\n",
2265             "64", r);
2266         retval = -EMSGSIZE;
2267         continue;
2268     }
2269 #undef GET_DESCRIPTOR_BUFSIZE
2270 }
2271
2272     for (j = 0; j < SET_ADDRESS_TRIES; ++j) {
2273         retval = hub_set_address(udev);
2274         if (retval >= 0)
2275             break;
2276         msleep(200);
2277     }
2278     if (retval < 0) {
2279         dev_err(&udev->dev,
2280             "device not accepting address %d, error %d\n",
2281             udev->devnum, retval);
2282         goto fail;
2283     }
2284
2285     /* cope with hardware quirkiness:
2286     * - let SET_ADDRESS settle, some device hardware wants it
2287     * - read ep0 maxpacket even for high and low speed,
2288     */
2289     msleep(10);
2290     if (USE_NEW_SCHEME(retry_counter))
2291         break;
2292
2293     retval = usb_get_device_descriptor(udev, 8);
2294     if (retval < 8) {
2295         dev_err(&udev->dev, "device descriptor "
2296             "read/%s, error %d\n",
2297             "8", retval);
2298     }
2299     if (retval >= 0)
2300         retval = -EMSGSIZE;
2301     } else {
2302         retval = 0;

```

```

2302         break;
2303     }
2304 }
2305 if (retval)
2306     goto fail;
2307
2308 i = udev->descriptor.bMaxPacketSize0 == 0xff?
2309     512 : udev->descriptor.bMaxPacketSize0;
2310 if (le16_to_cpu(udev->ep0.desc.wMaxPacketSize) != i) {
2311     if (udev->speed != USB_SPEED_FULL ||
2312         !(i == 8 || i == 16 || i == 32 || i == 64)) {
2313         dev_err(&udev->dev, "ep0 maxpacket = %d\n", i);
2314         retval = -EMSGSIZE;
2315         goto fail;
2316     }
2317     dev_dbg(&udev->dev, "ep0 maxpacket = %d\n", i);
2318     udev->ep0.desc.wMaxPacketSize = cpu_to_le16(i);
2319     ep0_reinit(udev);
2320 }
2321
2322 retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);
2323 if (retval < (signed)sizeof(udev->descriptor)) {
2324     dev_err(&udev->dev, "device descriptor read/%s, error %d\n",
2325         "all", retval);
2326     if (retval >= 0)
2327         retval = -ENMSG;
2328     goto fail;
2329 }
2330
2331 retval = 0;
2332
2333 fail:
2334     if (retval)
2335         hub_port_disable(hub, port1, 0);
2336     mutex_unlock(&usb_address0_mutex);
2337     return retval;
2338 }

```

hub\_port\_init()这个函数的基本思想就是做初始化，首先是把一个设备 reset，然后是分配地址。接着获得设备描述符。

首先 DEFINE\_MUTEX 是来自于 include/linux/mutex.h 中的一个宏，用它可以定义一把互斥锁，在 Linux 内核中，其实是在 2005 年底才建立比较系统、完善的互斥锁机制，在那年冬天，来自 RedHat 公司的 Ingo Molnar 大胆地提出了他所谓的 Generic Mutex Subsystem，即通用的互斥锁机制。此前内核中很多地方使用的都是信号量，而当时间的箭头指向了 2005 年末时，“区里”（开源社区，下称区里）很多人抱怨说信号量不灵，很多时候不好用，当然“区里”为这事展开了一场轰轰烈烈的讨论。

老黑客 Ingo Molnar 受不了了，在一周之后，愤然提出要对内核进行一场大的革命。当时 Ingo 提出了诸多理由要求使用新的互斥锁机制，而不是过去普遍出现在内核中的信号量机制，比如新的机制占用更小的内存，代码更为紧凑，更快，更便于调试。在诸多优势的诱惑下，“区里”的成员将信将疑地便认可了这种做法。



忽如一夜春风来，在紧接着的几个里，人们纷纷提交 patch，把原来用信号量的地方都改成了互斥锁。而这种改变深入到 Linux 中 USB 子系统是始于 2006 年春天 Greg 将 USB 中的代码中绝大多数的信号量代码换成了互斥锁代码。所以到今天，在 2.6.22 内核的代码中，整个 USB 子系统里几乎没有了 down/up 这一对函数的使用，取而代之的是 mutex\_lock() 和 mutex\_unlock() 函数对。而要初始化，只需像我们这里一样，DEFINE\_MUTEX(name) 即可。关于这个新的互斥锁的定义在 include/linux/mutex.h 中，而实现在 kernel/mutex.c 中。

我们继续往下看。hdev 被定义用来记录 Hub 所对应的 struct usb\_device，而 delay 记录延时，因为 USB 设备的 reset 工作不可能是瞬间完成的，通常会有一点点延时，这很容易理解，计算机永远不可能说你按一下 reset 键就立刻能够重新启动并马上就能重新工作的，这里首先给 delay 设置的初始值为 10 ms，即 HUB\_SHORT\_RESET\_TIME 这个宏被定义为 10 ms。这个 10 ms 的来源是 spec 中 7.1.7.5 节 Reset Signaling 里面说的，“The reset signaling must be driven for a mininum of 10 ms”，这个 10 ms 在 spec 中称之为 TDRST。一个 Hub 端口在 reset 之后将进入 Enabled 状态。

然后定义一个 oldspeed 用来记录设备在没有 reset 之前的速度。

2120 行，只有 Root Hub 没有父亲，而 spec 里说得很清楚，“It is required that resets from root ports have a duration of at least 50 ms”，这个 50 ms 被称为 TDRSTR。即 HUB\_ROOT\_RESET\_TIME 这个宏被定义为 50 ms。

2122 和 2123 行是 OTG 相关的，HNP 是 OTG 标准所支持的协议，HNP 即 Host Negotiation Protocol，俗称主机通令协议。咱们既然不关注 OTG，那么这里也就不做解释了。省得把问题复杂化了。

2128 行，这两行代码来源于实践。实践表明，某些低速设备要求有比较高的延时才能完成好它们的 reset，这很简单，286 的机器重启肯定比奔腾 4 的机器要慢。

调用 mutex\_lock 获得互斥锁，即表明下面这段代码一个时间只能被一个进程执行。然后调用 hub\_port\_reset()。

```
1509 static int hub_port_reset(struct usb_hub *hub, int port1,
1510                          struct usb_device *udev, unsigned int delay)
1511 {
1512     int i, status;
1513
1514     /* Reset the port */
1515     for (i = 0; i < PORT_RESET_TRIES; i++) {
1516         status = set_port_feature(hub->hdev,
1517                                 port1, USB_PORT_FEAT_RESET);
1518         if (status)
1519             dev_err(hub->intfdev,
1520                   "cannot reset port %d (err = %d)\n",
1521                   port1, status);
1522     } else {
```

```

1523         status = hub_port_wait_reset(hub, port1, udev, delay);
1524         if (status && status != -ENOTCONN)
1525             dev_dbg(hub->intfdev,
1526                     "port_wait_reset: err = %d\n",
1527                     status);
1528     }
1529
1530     /* return on disconnect or reset */
1531     switch (status) {
1532     case 0:
1533         /* TRSTRCY = 10 ms; plus some extra */
1534         msleep(10 + 40);
1535         /* FALL THROUGH */
1536     case -ENOTCONN:
1537     case -ENODEV:
1538         clear_port_feature(hub->hdev,
1539                             port1, USB_PORT_FEAT_C_RESET);
1540         /* FIXME need disconnect() for NOTATTACHED device */
1541         usb_set_device_state(udev, status
1542                             ? USB_STATE_NOTATTACHED
1543                             : USB_STATE_DEFAULT);
1544         return status;
1545     }
1546
1547     dev_dbg (hub->intfdev,
1548             "port %d not enabled, trying reset again...\n",
1549             port1);
1550     delay = HUB_LONG_RESET_TIME;
1551 }
1552
1553 dev_err (hub->intfdev,
1554         "Cannot enable port %i. Maybe the USB cable is bad?\n",
1555         port1);
1556
1557 return status;
1558 }

```

事到如今，有些函数不讲也不行了。这就是 `set_port_feature` 函数。其实之前我们遇见过，只是因为当时属于可讲可不讲，所以就先跳过去了。但现在不讲不行了，我们前面讲过它的搭档 `clear_port_feature`，所以我不讲你也应该知道 `set_port_feature()` 干什么用的。很显然，一个是清除 feature，一个是设置 feature。Linux 中很多这种成对的函数，刚才讲的那个 `mutex_lock()` 和 `mutex_unlock()` 不也是这样吗？

```

172 /*
173  * USB 2.0 spec Section 11.24.2.13
174  */
175 static int set_port_feature(struct usb_device *hdev, int port1, int feature)
176 {
177     return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
178                             USB_REQ_SET_FEATURE, USB_RT_PORT, feature, port1,
179                             NULL, 0, 1000);
180 }

```

看明白了 `clear_port_feature()` 的读者一定不会觉得这个函数难以理解。发送一个控制请求，设置一个 feature，这里传递进来的 feature 是 `USB_PORT_FEAT_RESET`，即对应于 spec 中的 reset。

发送好了之后就延时等待，调用 `hub_port_wait_reset()`:

```

1457 static int hub_port_wait_reset(struct usb_hub *hub, int port1,
1458                               struct usb_device *udev, unsigned int delay)
1459 {
1460     int delay_time, ret;
1461     u16 portstatus;
1462     u16 portchange;
1463
1464     for (delay_time = 0;
1465          delay_time < HUB_RESET_TIMEOUT;
1466          delay_time += delay) {
1467         /* wait to give the device a chance to reset */
1468         msleep(delay);
1469
1470         /* read and decode port status */
1471         ret = hub_port_status(hub, port1, &portstatus, &portchange);
1472         if (ret < 0)
1473             return ret;
1474
1475         /* Device went away? */
1476         if (!(portstatus & USB_PORT_STAT_CONNECTION))
1477             return -ENOTCONN;
1478
1479         /* bomb out completely if something weird happened */
1480         if ((portchange & USB_PORT_STAT_C_CONNECTION))
1481             return -EINVAL;
1482
1483         /* if we've finished resetting, then break out of the loop */
1484         if (!(portstatus & USB_PORT_STAT_RESET) &&
1485             (portstatus & USB_PORT_STAT_ENABLE)) {
1486             if (hub_is_wusb(hub))
1487                 udev->speed = USB_SPEED_VARIABLE;
1488             else if (portstatus & USB_PORT_STAT_HIGH_SPEED)
1489                 udev->speed = USB_SPEED_HIGH;
1490             else if (portstatus & USB_PORT_STAT_LOW_SPEED)
1491                 udev->speed = USB_SPEED_LOW;
1492             else
1493                 udev->speed = USB_SPEED_FULL;
1494             return 0;
1495         }
1496
1497         /* switch to the long delay after two short delay failures */
1498         if (delay_time >= 2 * HUB_SHORT_RESET_TIME)
1499             delay = HUB_LONG_RESET_TIME;
1500
1501         dev_dbg (hub->intfdev,
1502                 "port %d not reset yet, waiting %d ms\n",
1503                 port1, delay);
1504     }
1505
1506     return -EBUSY;
1507 }

```

这里 `HUB_RESET_TIMEOUT` 是设置的一个超时，这个宏的值为 500 毫秒，即如果“reset”了 500 毫秒还没好，那么就返回错误值。而循环的步长正是我们前面设置的那个 `delay`。

`msleep(delay)`就是休眠 `delay` 毫秒。

休眠完了就读取端口的状态。`hub_port_status()`不用说了，咱们前面讲过了。获得端口状态。错误就返回错误码，正确就把信息记录在 `portstatus` 和 `portchange` 里。

1476 行判断，如果在 `reset` 期间设备都被撤掉了，那就返回吧。

1480 行判断，如果又一次汇报说有设备插入，那就是返回错误。

正如刚才说过的，`reset` 真正完成以后，`status` 就应该是 `enabled`，所以 1484 行和 1485 行的 `if` 如果满足就说明 `reset` 好了。

1486 行的 `if` 是判断否是一个无线 Root Hub，即既是 Root Hub，又是无线 Hub，因为在 `struct usb_hcd` 中有一个成员 `unsigned wireless`，这个 `flag` 标志了该主机控制器是 `Wireless` 的。如果是 `Wireless` 的话，`speed` 就是 `USB_SPEED_VARIABLE`。

否则如果 `portstatus` 和 `USB_PORT_STAT_HIGH_SPEED` 相与结果为 1，则是高速设备，如果与 `USB_PORT_STAT_LOW_SPEED` 相与结果为 1 则是低速设备，剩下的可能就是全速设备。到这里，这个 `hub_port_wait_reset` 就可以返回了，正常返回值为 0。注意，经过这里 `udev` 的 `speed` 就被设置好了。

1497 行和 1498 行，进行到这里就说明还没有“`reset`”好。如果已经过了两个 `HUB_SHORT_RESET_TIME`，就把步长设置为 `HUB_LONG_RESET_TIME`，即 200 ms，然后打印一条警告信息，继续循环，如果循环完全结束还不行，那就说明超时了，返回 `-EBUSY`。

回到 `hub_port_reset`，下面走到了 1531 行，一个 `switch`，根据刚才的返回值做一次选择，如果结果为 0，说明正常，为了安全起见，索性再等 50 ms。如果是错误码，并且错误码表明设备不在了，则首先清掉 `reset` 这个 `feature`，然后把这个为刚才这个设备申请的 `struct usb_device` 结构体的状态设置为 `USB_STATE_NOTATTACHED` 并且返回 `status`。

在 `switch` 里面，如果 `status` 为 0，那么由于 `case 0` 那一部分后面的 `break` 语句，`case -ENOTCONN` 和 `case -ENODEV` 下面的那几句代码都会执行，即 `clear_port_feature` 是总会执行的，`usb_set_device_state()`也会执行，而对于 `status` 为 0 的情况，属于正常情况。从这时开始，`struct usb_device` 结构体的状态就将记录为 `USB_STATE_DEFAULT`，即所谓的默认状态，然后返回值就是 0。而对于端口 `reset`，我们设置的重复次数是 `PORT_RESET_TRIES`，它等于 5，你当然可以把它改为 1。只要你对自己的设备够自信，一次 `reset` 就肯定成功。

如果 1553 行还会执行，那么说明肯定出了大问题了，`reset` 都没法进行，于是返回错误状态。

回到 `hub_init_port()`中，如果刚才失败了，就 `goto fail`，否则也暂时将 `retval` 这个临时变量设置为 `-ENODEV`。而 2140 行，如果 `oldspeed` 不是 `USB_SPEED_UNKNOWN`，并且也不等于

刚刚设置的 speed, 那么说明 reset 之前设备已经在工作了, 而这次 reset 把设备原来的速度状态也给改变了。这是不合理的, 必须结束函数, 并且 disable 这个端口。于是 goto fail, 而在 fail 那边可以看到, 由于 retval 不为 0, 所以调用 hub\_port\_disable()关掉这个端口。然后释放互斥锁, 并且返回错误代码。

2144 行, 如果不是刚才这种情况, 那么令 oldspeed 等于现在这个 udev->speed。

2151 行开始, 又是一个 switch, 感觉这一段代码的选择出现得太多了。

其实这里是设置一个初始值, 我们曾经介绍过 ep0。ep0.desc.wMaxPacketSize 用来记录端点 0 的单个包的最大传输 size。对于无线设备, 无线 USB spec 规定了, 端点 0 的最大包 size 就是 512Bytes, 而对于高速设备, 这个也是 USB spec 规定好了, 即 64Bytes, 而低速设备同样是 USB spec 规定好了, 8Bytes。唯一存在变数的是全速设备, 它可能是 8Bytes, 可能是 16Bytes, 可能是 32Bytes, 也可能是 64Bytes, 对于这种设备, 只能通过读取设备描述符来获得了。正如我们说过的, Linux 向 Windows 妥协了, 这里全速的设备, 默认先把 wMaxPacketSize 设置为 64Bytes, 而不是以前的那种 8Bytes。

2172 行至 2186 行, 仅仅是为了打印一行调试信息, 对于写代码的人来说, 可能很有用, 而对读代码的人来说, 也许就没有任何意义了。

```
Sep  9 11:32:49 localhost kernel: usb 4-5: new high speed USB device using ehci_hcd
and address 3
Sep  9 11:32:52 localhost kernel: usb 2-1: new low speed USB device using uhci_hcd
and address 3
```

比如在我的计算机里, 就可以在 /var/log/messages 日志文件中看到上面这样的信息。ehci\_hcd 和 uhci\_hcd 就是主机控制器的驱动程序, 3 就是设备的 devnum。

2189 行, 不是 switch 而是 if, 除了判断还是判断, 只不过刚才是选择, 现在是如果, 这里如果 hdev->tt 为真, 则如何如何, 否则, 如果设备本身不是高速的, 而 hub 是高速的, 那么如何如何。tt 我们介绍过即 transaction translator, 而 ttport 是 struct usb\_device 中的一个成员, int ttport, 这个 ttport 以后会被用到, tt 也将在以后会被用到, 暂时先不细说, 到时候再回来看。

GET\_DESCRIPTOR\_TRIES 等于 2, 这段代码的思想我们以前就讲过, USB\_NEW\_SCHEME(retry\_counter), retry\_counter 就是 hub\_port\_init()传递进来的最后一个参数, 而我们给它的实参正是从 0 到 SET\_CONFIG\_TRIES-1 的 i。假设我们什么也没有设置, 都是使用默认值, 那么 use\_both\_schemes 默认值为 1, 而 old\_scheme\_first 默认值为 0, 于是 SET\_CONFIG\_TRIES 为 4, 即 i 将从 0 变到 3, 而 USB\_NEW\_SCHEME(i)将在 i 为 0 和 1 时为 1, 在 i 为 2 和 3 时为 0。所以, 先进行两次新的策略, 如果不行就再进行两次旧的策略。所有这一切只有一个目的, 就是为了获得设备的描述符。由于思想介绍已经非常清楚, 代码我们就不再一行一行讲了。尤其是那些错误判断的句子。

只是介绍一下其中调用的几个函数。对于新策略,首先定义一个 `struct usb_device_descriptor` 的指针 `buf`,然后申请 64 个字节的空間,发送一个控制传输的请求,结束之后,查看 `buf->bMaxPacketSize0`,合理值只有 8/16/32/64/512。这里的 255 实际上是 WUSB 协议规定的,毕竟只有 8 位,最大就是 255 了,所以就用这个值来代表 WUSB 设备。实际上 WUSB 的大小是 512。循环三次是为了保险起见,因为实践表明这类请求通常成功率很难达到 100%。

2249 行用 `udev->descriptor.bMaxPacketSize0` 来记录这个临时获得的值。然后 `buf` 的使命结束了,释放它的内存。

正如我们曾经分析的那样,把设备 `reset`。

接下来是设置地址, `hub_set_address()`:

```

2076 static int hub_set_address(struct usb_device *udev)
2077 {
2078     int retval;
2079
2080     if (udev->devnum == 0)
2081         return -EINVAL;
2082     if (udev->state == USB_STATE_ADDRESS)
2083         return 0;
2084     if (udev->state != USB_STATE_DEFAULT)
2085         return -EINVAL;
2086     retval = usb_control_msg(udev, usb_sndaddr0pipe(),
2087                             USB_REQ_SET_ADDRESS, 0, udev->devnum, 0,
2088                             NULL, 0, USB_CTRL_SET_TIMEOUT);
2089     if (retval == 0) {
2090         usb_set_device_state(udev, USB_STATE_ADDRESS);
2091         ep0_reinit(udev);
2092     }
2093     return retval;
2094 }
```

和前面的 `choose_address` 不同, `choose_address` 是从软件意义上挑选一个地址。而这里要发送真正的请求,因为设置设备地址本身就是 `spec` 规定的标准的请求之一,这里我们用宏 `USB_REQ_SET_ADDRESS` 来代替,只有真正发送了请求之后,硬件上才能真正通过这个地址进行通信。这里最关键的就是传递了 `udev->devnum`,这正是我们前面选择的地址,这里赋予了 `wIndex`。

设好了地址之后,把设备的状态从开始的 `USB_STATE_DEFAULT` 变成了 `USB_STATE_ADDRESS`。在 `spec` 中,这个状态被称为 Address state,我叫它有地址的状态。

然后调用了 `ep0_reinit()`。

```

2066 static void ep0_reinit(struct usb_device *udev)
2067 {
2068     usb_disable_endpoint(udev, 0 + USB_DIR_IN);
2069     usb_disable_endpoint(udev, 0 + USB_DIR_OUT);
2070     udev->ep_in[0] = udev->ep_out[0] = &udev->ep0;
2071 }
```

`usb_disable_endpoint` 是 `usbcore` 提供的一个函数，具体到这个端点，它会让 `ep_out[0]` 和 `ep_in[0]` 置为 `NULL`。接着会取消掉任何挂在端点上的 `urb`。这里 2070 行再次把 `ep_in[0]` 和 `ep_out[0]` 指向 `udev->ep0`。`ep0` 是真正占内存的数据结构，而 `ep_in[0]` 和 `ep_out[0]` 只是指针，而在主机控制器驱动程序里将被用到的正是这两个指针数组。

你也许会问，这里为何要调用 `ep0_reinit()` 函数？而且在别的地方也有看见调用 `ep0_reinit()` 函数的？首先，主机控制器通常会记录着每一个端点的状态，（否则它怎么知道如何跟每个端点通信？）而这个东西在每次设备状态发生了变化了之后就要相应地作出变化，或者说 *needs to be cleared out*。

具体来说，`ep0` 是一个 `struct usb_host_endpoint` 的结构体，这个结构体的变量都是为主机控制器驱动来使用的。`struct usb_host_endpoint` 里有一个成员 `struct list_head urb_list`，也就是说，所有针对该端点的 `urb` 请求被排列成一个队列，在我们这个上下文里，也许我们的设备还没有任何 `urb` 请求，但是要知道 `hub_set_address()` 这个函数是被 `hub_port_init` 调用，而 `hub_port_init()` 并不是只有在这个上下文里被调用，也许下一次调用时，`ep0` 对应的 `urb_list` 里面已经有东西了，那么这里重新设置一下地址，毫无疑问，需要把原来的那些 `urb` 请求给清除掉。

而这，正是 `ep0_reinit()` 所做的，它会通知主机控制器，由主机控制器来具体实施。

当一个设备没有被设置地址时，它使用默认地址，即地址 0，而当我们设置地址之后，这个地址发生了变化，所以主机控制器必须知道这件事情，否则它没法控制。同样，这里 `usb_disable_endpoint()` 最终会调用主机控制器驱动提供的函数，让主机控制器驱动作出相应的反应。

回到 `hub_port_init` 中来，设置好了地址，然后 2289 行，先休眠 10 ms，然后如果还是新策略，那么就可以结束了，因为该做的都做完了。如果是旧策略，那么执行到这里还刚上路呢，我们说过了，这里的思路就是先获得设备描述符的前 8 个字节，从中得知 `udev->descriptor` 的 `bMaxPacketSize0`，然后再次调用 `usb_get_device_descriptor` 完整的获得一次设备描述符。到 2336 行，释放互斥锁。

至此，`hub_port_init()` 就可以返回了。

声明一下，`usb_get_device_descriptor()` 是来自 `drivers/usb/core/message.c` 中的函数，由 `usbcore` 提供，我们这里就不细讲了。其作用是获取设备描述符。

这里 2319 行又调用了一次 `ep0_reinit`，理由很简单，这是针对全速设备而言的，因为别的 speed 的设备的 `ep0` 的最大包 size 一开始就设置好了，不会改变，而全速设备则是刚刚从设备描述符里读出来，而且 2310 行的意思是读出来这个值和最初我们设想的那个值不同，当然以读出来的为准，所以 `ep0.desc.wMaxpacketSize` 又有变化，有了变化就得调用 `ep0_reinit` 让主机控制器驱动知道，与前面换地址的情况类似，只不过这次是换包的大小，自然也需要把 `urb` 给清掉。

## 21. 八大重量级函数闪亮登场（四）

2514 行至 2536 行，整个这一块代码是专门为了处理 Hub 的，2514 行这个 if 语句是判断，接在当前 Hub 端口的设备不是别的普通设备，恰恰也正是另一个 Hub，即所谓的级联。

udev->bus\_mA 刚刚在 2493 行设置的，即 hub 能够提供的每个端口的电流。在 hub\_configure 函数中面我们曾经设置了 hub->mA\_per\_port，如果它小于等于 100mA，说明设备供电是存在问题的，电力不足。那么这种情况我们首先要判断接入的这个 Hub 是不是也得靠总线供电，如果是，那就麻烦了。所以这里再次调用 usb\_get\_status() 函数，虽说我们把这个函数列入八大重量级函数之一，但是实际上我们前面已经讲过这个函数了，所以这里不必进入函数内部，只是需要知道 usb\_get\_status 一执行，正常的话，这个子 hub 的状态就被记录在 devstat 里面了，而 2525 行的意思是如果这个设备不能执行，那么我们就打印一条错误信息，然后 goto loop\_disable，关闭这个端口，结束这次循环。

2529 行至 2533 行又是指示灯相关的代码，我们在前面已经讲过了，此刻 schedule\_delayed\_work(&hub->leds, 0) 函数一执行，就意味着当初注册的 led\_work() 函数将会立刻被调用。这个函数其实挺简单的，代码虽然不短，但是都是简单的代码。考虑到 usb\_get\_status() 我们不用讲了，所以这里就干脆顺便看一下这个简单函数吧。

```

208 #define LED_CYCLE_PERIOD      ((2*HZ)/3)
209
210 static void led_work (struct work_struct *work)
211 {
212     struct usb_hub          *hub =
213         container_of(work, struct usb_hub, leds.work);
214     struct usb_device        *hdev = hub->hdev;
215     unsigned                 i;
216     unsigned                 changed = 0;
217     int                      cursor = -1;
218
219     if (hdev->state != USB_STATE_CONFIGURED || hub->quiescing)
220         return;
221
222     for (i = 0; i < hub->descriptor->bNbrPorts; i++) {
223         unsigned             selector, mode;
224
225         /* 30%-50% duty cycle */
226
227         switch (hub->indicator[i]) {
228             /* cycle marker */
229             case INDICATOR_CYCLE:
230                 cursor = i;
231                 selector = HUB_LED_AUTO;
232                 mode = INDICATOR_AUTO;
233                 break;
234             /* blinking green = sw attention */
235             case INDICATOR_GREEN_BLINK:
236                 selector = HUB_LED_GREEN;
237                 mode = INDICATOR_GREEN_BLINK_OFF;

```



```

238         break;
239     case INDICATOR_GREEN_BLINK_OFF:
240         selector = HUB_LED_OFF;
241         mode = INDICATOR_GREEN_BLINK;
242         break;
243     /* blinking amber = hw attention */
244     case INDICATOR_AMBER_BLINK:
245         selector = HUB_LED_AMBER;
246         mode = INDICATOR_AMBER_BLINK_OFF;
247         break;
248     case INDICATOR_AMBER_BLINK_OFF:
249         selector = HUB_LED_OFF;
250         mode = INDICATOR_AMBER_BLINK;
251         break;
252     /* blink green/amber = reserved */
253     case INDICATOR_ALT_BLINK:
254         selector = HUB_LED_GREEN;
255         mode = INDICATOR_ALT_BLINK_OFF;
256         break;
257     case INDICATOR_ALT_BLINK_OFF:
258         selector = HUB_LED_AMBER;
259         mode = INDICATOR_ALT_BLINK;
260         break;
261     default:
262         continue;
263     }
264     if (selector != HUB_LED_AUTO)
265         changed = 1;
266     set_port_led(hub, i + 1, selector);
267     hub->indicator[i] = mode;
268 }
269 if (!changed && blinkenlights) {
270     cursor++;
271     cursor %= hub->descriptor->bNbrPorts;
272     set_port_led(hub, cursor + 1, HUB_LED_GREEN);
273     hub->indicator[cursor] = INDICATOR_CYCLE;
274     changed++;
275 }
276 if (changed)
277     schedule_delayed_work(&hub->leds, LED_CYCLE_PERIOD);
278 }

```

注意了，刚才进入这个函数之前，我们设置了 `hub->indicator[port1-1]` 为 `INDICATOR_AMBER_BLINK`，而眼下这个函数从 222 行开始主循环，有多少个端口就循环多少次，即遍历端口。227 行进行判断，对于这个情形，很显然，`selector` 被设置为 `HUB_LED_AMBER`，这个宏的值为 1。而 `mode` 被设置为 `INDICATOR_AMBER_BLINK_OFF`。

然后 266 行，`set_port_led()`：

```

182 /*
183  * USB 2.0 spec Section 11.24.2.7.1.10 and table 11-7
184  * for info about using port indicators
185  */
186 static void set_port_led(
187     struct usb_hub *hub,
188     int port1,

```

```

189     int selector
190 )
191 {
192     int status = set_port_feature(hub->hdev, (selector << 8) | port1,
193                                   USB_PORT_FEAT_INDICATOR);
194     if (status < 0)
195         dev_dbg (hub->intfdev,
196                 "port %d indicator %s status %d\n",
197                 port1,
198                 ({ char *s; switch (selector) {
199                     case HUB_LED_AMBER: s = "amber"; break;
200                     case HUB_LED_GREEN: s = "green"; break;
201                     case HUB_LED_OFF: s = "off"; break;
202                     case HUB_LED_AUTO: s = "auto"; break;
203                     default: s = "??"; break;
204                 }; s; })),
205                 status);
206 }

```

看到调用 `set_port_feature` 我们就熟悉了，`USB_PORT_FEAT_INDICATOR` 对应 spec 中的 `PORT_INDICATOR` 这个特征，参看 spec 的 Table 11-25。

这里传递进来的是 Amber，即 Selector 为 1，于是指示灯会亮 Amber，即琥珀色。这里我们看到有两个 Mode，一个是 Automatic，一个是 Manual，Automatic 就是灯自动闪，自动变化，而 Manual 基本上就是说我们需要用软件来控制灯的闪烁。我们选择的是后者，所以 265 行我们就设置 `changed` 为 1。这样，走到 276 行，`schedule_delayed_work()` 再次执行，但这次执行时，第二个参数不再是 0，而是 `LED_CYCLE_PERIOD`，即 0.66HZ。而我们现在的 `hub->indicator[port1-1]` 和刚才进来时相反，是 `INDICATOR_AMBER_BLINK_OFF`，于是你会发现下次再进来又会变成 `INDICATOR_AMBER_BLINK`，如此反复。这就意味着，这个函数接下来将以这个频率被调用，也就是说指示灯将以 0.66HZ 的频率进行开关。

不过需要说明的是，注意我们刚才是如何进入到这个函数的，是因为遇见了电力不足的情况进来的，所以这并不是什么好事，通常琥珀色的灯亮了话，说明硬件方面有问题。按 spec 规定，指示灯是琥珀色，亮而不闪，表明是错误环境；指示灯是琥珀色，又亮又闪，表明硬件有问题，只有指示灯是绿色才是工作状态；如果指示灯是绿色，闪烁，那说明软件有问题。

接下来我们把第六个函数也讲了。`check_highspeed()`，看了名字基本就知道是什么意思了。spec 里面规定，一个设备如果能够进行高速传输，那么它就应该在设备描述符里的 `bcdUSB` 这一项写上 0200H。`highspeed_hubs` 这个变量干什么的？`drivers/usb/core/hub.c` 中定义的一个静态变量。不要说你是第一次见它，在 `hub_probe()` 里面就和它见过。让我们把思绪拉回到 `hub_probe` 中去，当时我们就判断了如果 `hdev->speed` 是 `USB_SPEED_HIGH`，则 `highspeed_hubs++`，所以现在这里的意思就很明确了，如果有高速的 Hub，而你又能进行高速传输，还要进行全速传输。这种情况下，调用 `check_highspeed()`。

```

2340 static void
2341 check_highspeed (struct usb_hub *hub, struct usb_device *udev, int port1)
2342 {

```

```

2343     struct usb_qualifier_descriptor *qual;
2344     int                               status;
2345
2346     qual = kmalloc (sizeof *qual, GFP_KERNEL);
2347     if (qual == NULL)
2348         return;
2349
2350     status = usb_get_descriptor (udev, USB_DT_DEVICE_QUALIFIER, 0,
2351                                qual, sizeof *qual);
2352     if (status == sizeof *qual) {
2353         dev_info(&udev->dev, "not running at top speed; "
2354                  "connect to a high speed hub\n");
2355         /* hub LEDs are probably harder to miss than syslog */
2356         if (hub->has_indicators) {
2357             hub->indicator[port1-1] = INDICATOR_GREEN_BLINK;
2358             schedule_delayed_work (&hub->leds, 0);
2359         }
2360     }
2361     kfree(qual);
2362 }

```

`struct usb_qualifier_descriptor` 这个结构体牵出了另外一个概念，Device Qualifier 描述符。首先这个结构体定义于 `include/linux/usb/ch9.h` 中：

```

348 /* USB_DT_DEVICE_QUALIFIER: Device Qualifier descriptor */
349 struct usb_qualifier_descriptor {
350     __u8  bLength;
351     __u8  bDescriptorType;
352
353     __le16 bcdUSB;
354     __u8  bDeviceClass;
355     __u8  bDeviceSubClass;
356     __u8  bDeviceProtocol;
357     __u8  bMaxPacketSize0;
358     __u8  bNumConfigurations;
359     __u8  bRESERVED;
360 } __attribute__((packed));

```

当我们设计 USB 2.0 时务必要考虑与过去的 USB 1.1 的兼容，高速设备如果接在一个旧的 Hub 上面，总不能说用不了吧？所以，如今的高速设备通常是可以以高速的方式也可以不以高速的方式工作，即可以调节，接在高速 Hub 上就按高速工作，如果不然，那么就按全速的方式去工作。

那么这一点是如何实现的呢？首先，在高速和全速下有不同设备配置信息的高速设备必须具有一个 `device_qualifier` 描述符，你可以叫它设备限定符描述符，不过这个叫法过于别扭，所以我们直接用英文，就叫 Device Qualifier 描述符。它干什么用的呢？它描述了一个高速设备在进行速度切换时所需改变的信息。比如，一个设备当前工作处于全速状态，那么 Device Qualifier 描述符中就保存着信息记录这个设备工作在高速状态的信息，反之如果一个设备当前工作于高速状态，那么 Device Qualifier 描述符中就包含着这个设备工作于全速状态的信息。

这里我们看到，首先定义一个 Device Qualifier 描述符的指针 `qual`，为其申请内存空间，然

后 `usb_get_descriptor` 去获得这个描述符，这个函数的返回值就是设备返回了多少个 Bytes。如果的确是 Device Qualifier 描述符的大小，那么说明这个设备的确是可以工作在高速状态的，因为全速设备是没有 Device Qualifier 描述符的，只有具有高速工作能力的设备才具有 Device Qualifier 描述符。而对于全速设备，在收到这个请求之后，返回的只是错误码。所以这里的意思就是说如果你这个设备确实是能够高速工作的，然而你却偏偏全速工作，并且我们刚才调用 `check_highspeed()` 之前也看到了，我们已经判断出设备是工作于全速而系统里有高速的 hub，那么至少这说明这个设备不正常，所以剩下的代码就和刚才那个指示灯是闪烁琥珀色的道理一样，只不过这次是指示灯是闪绿灯，通常闪绿灯的意思是出现软件问题。

好了，又讲完一个函数。至此我们已经过五关斩六将，八个函数讲完了六个。你也许觉得这些函数很枯燥，觉得读代码很辛苦，不过很不幸，我得告诉你，其实真正最重要的函数是第七个，即 `usb_new_device()`，这个函数一结束你就可以用 `lsusb` 命令看到你的设备了。

## 22. 八大重量级函数闪亮登场（五）

在调用 `usb_new_device` 之前，如果 Hub 已经北撤掉了，2555 行到 2560 行这段代码就忽略了。否则，把 `udev` 赋值给 `hdev->children` 数组中的对应元素，也正是从此以后，这个设备才算是真正挂上了这棵大树。

如果 `status` 确实为 0，（注意，2549 刚刚把 `status` 赋为了 0。）正式调用 `usb_new_device`。

```

1295 int usb_new_device(struct usb_device *udev)
1296 {
1297     int err;
1298
1299     /* Determine quirks */
1300     usb_detect_quirks(udev);
1301
1302     err = usb_get_configuration(udev);
1303     if (err < 0) {
1304         dev_err(&udev->dev, "can't read configurations, error %d\n",
1305             err);
1306         goto fail;
1307     }
1308
1309     /* read the standard strings and cache them if present */
1310     udev->product = usb_cache_string(udev, udev->descriptor.iProduct);
1311     udev->manufacturer = usb_cache_string(udev,
1312         udev->descriptor.iManufacturer);
1313     udev->serial = usb_cache_string(udev,
1314         udev->descriptor.iSerialNumber);
1315
1316     /* Tell the world! */
1317     dev_dbg(&udev->dev, "new device strings: Mfr=%d, Product=%d, "
1318         "SerialNumber=%d\n",

```

```

1318         udev->descriptor.iManufacturer,
1319         udev->descriptor.iProduct,
1320         udev->descriptor.iSerialNumber);
1321     show_string(udev, "Product", udev->product);
1322     show_string(udev, "Manufacturer", udev->manufacturer);
1323     show_string(udev, "SerialNumber", udev->serial);
1324
1325 #ifdef CONFIG_USB_OTG
1326     /*
1327     * OTG-aware devices on OTG-capable root hubs may be able to use SRP,
1328     * to wake us after we've powered off VBUS; and HNP, switching roles
1329     * "host" to "peripheral". The OTG descriptor helps figure this out.
1330     */
1331     if (!udev->bus->is_b_host
1332         && udev->config
1333         && udev->parent == udev->bus->root_hub) {
1334         struct usb_otg_descriptor *desc = 0;
1335         struct usb_bus *bus = udev->bus;
1336
1337         /* descriptor may appear anywhere in config */
1338         if (__usb_get_extra_descriptor (udev->rawdescriptors[0],
1339             le16_to_cpu(udev->config[0].desc.wTotalLength),
1340             USB_DT_OTG, (void **) &desc) == 0) {
1341             if (desc->bmAttributes & USB_OTG_HNP) {
1342                 unsigned port1 = udev->portnum;
1343
1344                 dev_info(&udev->dev,
1345                     "Dual-Role OTG device on %sHNP port\n",
1346                     (port1 == bus->otg_port)
1347                     ? "" : "non-");
1348
1349                 /* enable HNP before suspend, it's simpler */
1350                 if (port1 == bus->otg_port)
1351                     bus->b_hnp_enable = 1;
1352                 err = usb_control_msg(udev,
1353                     usb_sndctrlpipe(udev, 0),
1354                     USB_REQ_SET_FEATURE, 0,
1355                     bus->b_hnp_enable
1356                     ? USB_DEVICE_B_HNP_ENABLE
1357                     : USB_DEVICE_A_ALT_HNP_SUPPORT,
1358                     0, NULL, 0, USB_CTRL_SET_TIMEOUT);
1359                 if (err < 0) {
1360                     /* OTG MESSAGE: report errors here,
1361                     * customize to match your product.
1362                     */
1363                     dev_info(&udev->dev,
1364                         "can't set HNP mode; %d\n",
1365                         err);
1366                     bus->b_hnp_enable = 0;
1367                 }
1368             }
1369         }
1370     }
1371
1372     if (!is_targeted(udev)) {
1373
1374         /* Maybe it can talk to us, though we can't talk to it.
1375         * (Includes HNP test device.)
1376         */

```

```

1377     if (udev->bus->b_hnp_enable || udev->bus->is_b_host) {
1378         err = __usb_port_suspend(udev, udev->bus->otg_port);
1379         if (err < 0)
1380             dev_dbg(&udev->dev, "HNP fail, %d\n", err);
1381     }
1382     err = -ENODEV;
1383     goto fail;
1384 }
1385 #endif
1386
1387 /* export the usbdev device-node for libusb */
1388 udev->dev.devt = MKDEV(USB_DEVICE_MAJOR,
1389                      (((udev->bus->busnum-1) * 128) + (udev->devnum-1)));
1390
1391 /* Register the device. The device driver is responsible
1392  * for adding the device files to sysfs and for configuring
1393  * the device.
1394  */
1395 err = device_add(&udev->dev);
1396 if (err) {
1397     dev_err(&udev->dev, "can't device_add, error %d\n", err);
1398     goto fail;
1399 }
1400
1401 /* Increment the parent's count of unsuspended children */
1402 if (udev->parent)
1403     usb_autoresume_device(udev->parent);
1404
1405 exit:
1406     return err;
1407
1408 fail:
1409     usb_set_device_state(udev, USB_STATE_NOTATTACHED);
1410     goto exit;
1411 }

```

这个函数看似很长，实则不然。幸亏在前面作了一个假设，即假设不打开支持 OTG 的代码。而剩下的代码就相对来说简单多了，主要就是调用了几个函数。下面一个一个来看。

usb\_detect\_quirks(), 好的 USB 设备都是相似的，大家遵守同样的游戏规则，而不好的 USB 设备却各有各的毛病。这里使用到两个文件，drivers/usb/core/quirks.c 及 include/linux/usb/quirks.h。quirk，即与众不同的意思。

在 include/linux/usb/quirks.h 中，我们看到这个文件超级短，只有两行有意义，其余的几行是注释：

```

1 /*
2  * This file holds the definitions of quirks found in USB devices.
3  * Only quirks that affect the whole device, not an interface,
4  * belong here.
5  */
6
7 /* device must not be autosuspended */
8 #define USB_QUIRK_NO_AUTOSUSPEND        0x00000001
9

```

```
10 /* string descriptors must not be fetched using a 255-Byte read */
11 #define USB_QUIRK_STRING_FETCH_255      0x00000002
```

这个文件总共是 11 行，而其中定义了两个 flag，第一个 USB\_QUIRK\_NO\_AUTOSUSPEND 表明这个设备不能自动挂起，执行自动挂起会对设备造成伤害，确切地说是设备会被 crash。而第二个宏，USB\_QUIRK\_STRING\_FETCH\_255，是说该设备在获取字符串描述符时会 crash。

与此同时，在 drivers/usb/core/quirks.c 中定义了下面这张表：

```
30 static const struct usb_device_id usb_quirk_list[] = {
31     /* HP 5300/5370C scanner */
32     { USB_DEVICE(0x03f0, 0x0701), .driver_info = USB_QUIRK_STRING_FETCH_255 },
33     /* Seiko Epson Corp - Perfection 1670 */
34     { USB_DEVICE(0x04b8, 0x011f), .driver_info =
USB_QUIRK_NO_AUTOSUSPEND },
35     /* Elsa MicroLink 56k (V.250) */
36     { USB_DEVICE(0x05cc, 0x2267), .driver_info =
USB_QUIRK_NO_AUTOSUSPEND },
37
38     { } /* terminating entry must be last */
39 };
```

这张表被称作 USB 黑名单。在 2.6.22 的内核中这张表里只记录了 3 个设备，但之所以创建这张表，目的在于将来可以扩充，比如又往这张表里添加了几个扫描仪，比如明基的 S2W 3300U，精工爱普生的 Perfection 1200，以及另几家公司的一些产品。所以 2.6.23 的内核中将会看到这张表的内容比现在丰富。而从原理上来说，usb\_detect\_quirks() 函数就是为了判断一个设备是不是在这张黑名单上，然后如果是，就判断它具体是属于哪种问题。

2007 年之后的内核中，struct usb\_device 结构体有一个元素 u32 quirks，就是用来做这个检测的，usb\_detect\_quirks 会为在黑名单中找得到的设备的 struct usb\_device 结构体中的 quirks 赋值，然后接下来相关的代码就会判断一个设备的 quirks 中的某一位是否设置了，目前 quirks 里面只有两位可以设置，即 USB\_QUIRKS\_STRING\_FETCH\_255 所对应的 0x00000002 和 USB\_QUIRK\_NO\_AUTOSUSPEND 所对应的 0x00000001。

1302 行，usb\_get\_configuration()，获得配置描述符，我想你如果清楚了如何获得设备描述符，自然就不难知道如何获得配置描述符，知道了配置描述符，自然就不难知道如何获得接口描述符，然后是端点描述符。

usb\_get\_configuration 来自 drivers/usb/core/config.c，举一个例子，我们知道一部手机可以有多种配置，比如可以摄像，可以接在电脑里当做一个 U 盘，那么这两种情况就属于不同的配置，在手机里面有相应的选择菜单，你选择了哪种它就按哪种配置进行工作，供你选择的这个功能就叫做配置。很显然，当你摄像时你不可以访问这块 U 盘，当你访问这块 U 盘时你不可以摄像，因为你做了选择。

另外，既然一个配置代表一种不同的功能，那么很显然，不同的配置可能需要的接口就不

一样，假设你的手机里从硬件上来说一共有 5 个接口，那么可能当你配置成 U 盘时它只需要用到某一个接口，当你配置成摄像时，它可能只需要用到另外两个接口，可能你还有别的配置，然后你可能就会用到剩下那两个接口，那么当你选择好一种配置之后，你给设备发送请求，请求去获得配置描述符时，设备返回给你的就绝不仅仅是一个配置描述符，它还必须返回更多的信息。按 spec 的说法就是，设备将返回的是除了配置描述符以外，与这种配置相关的接口描述符，以及与这些接口相关的端点描述符，都会一次性返回给你。

另外一点我需要提示的是，一个接口可以有多种设置，比如在打印机驱动程序里，不同的设置可以表明使用不同的通信协议，又比如在声音设备驱动中设置可以决定不同的音频格式。那么作为 USB 设备驱动程序我如何知道这些呢？首先，对于任何一个接口来说，spec 规定了默认的设置是设置 0，即 0 号设置是默认设置，而如果一个接口可以有多种设置，那么每一个设置将对应一个接口描述符，换言之，即便你只有一个接口，但是由于可能有两种设置，那么就有两个接口描述符，而它们对应于同一个接口编号，或者说我们知道接口描述符里面有一个成员，bInterfaceNumber 和一个 bAlternateSetting，就是对于这种情况，两个接口描述符将具有相同的 bInterfaceNumber；而不相同的是 bAlternateSetting，另一方面，因为不同的设置完全有可能导致需要不同的端点，所以也将有不同的端点描述符。

而总的来说，在我们的 USB 设备驱动程序可以正常工作之前，我们需要知道的信息是，接口描述符、设置，以及端点描述符，而这一切，都在设备中，我们所需要做的就是发送请求，然后设备就把相关信息返回给我们，我们就记录下来，填充好我们自己的数据结构。而这些数据结构，对所有的 USB 设备都是一样的，因为这些都是 spec 里面规定的，也正是因为如此，写代码的人们才把这部分工作交给 USB Core 来完成，而不是纷纷下放给每一个设备单独去执行，因为那样就太浪费时间了，大家都得干一些重复的工作，显然是没有必要的。

关于 usb\_get\_configuration() 函数我们就说这么多，我们传递的是 struct usb\_device 结构体指针，即这个故事中的 udev，从此以后你就会发现 udev 中的各个成员就有值了，这些值从哪来的？正是从设备中来。

回到 usb\_new\_device 中来，1310 行到 1323 行，还记得我们说过那个字符串描述符吧，这里就是去获得字符串描述符，并且保存下来。知道为什么用 lsusb 命令可以看到诸如下面的内容了吧。

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # lsusb
Bus 001 Device 001: ID 0000:0000
Bus 002 Device 003: ID 0624:0294 Avocent Corp.
Bus 002 Device 001: ID 0000:0000
Bus 003 Device 001: ID 0000:0000
Bus 004 Device 003: ID 04b4:6560 Cypress Semiconductor Corp. CY7C65640 USB-2.0
"TetraHub"
Bus 004 Device 001: ID 0000:0000
```

其中那些字符串，就是这里保存起来的。试想如果不保存起来，那么每次你执行 lsusb，都



要去向设备发送一次请求，那设备还不被烦死？`usb_cache_string()`就是干这个的，它来自 `drivers/usb/core/message.c`，从此 `udev->product`，`udev->manufacturer`，`udev->serial` 里面就有值了。而下面那几行就是打印出来，`show_string` 其实就是变相的 `printk` 语句。

接下来，我们已经说过了，OTG 的代码我们只能略过，然后就到了 1388 行，这里就是传统理论中的两个主设备号和次设备号了。按传统理论来说，主设备号表明了一类设备，一般对应着确定的驱动程序，而次设备号通常是因为一个驱动程序要支持多个设备而为了让驱动程序区分它们而设置的。比如我的硬盘，如下所示：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls -l /dev/sd*
brw-r----- 1 root disk 8,  0 Aug  6 18:19 /dev/sda
brw-r----- 1 root disk 8,  1 Aug  6 18:19 /dev/sda1
brw-r----- 1 root disk 8,  2 Aug  6 18:19 /dev/sda2
brw-r----- 1 root disk 8,  3 Aug  6 18:19 /dev/sda3
brw-r----- 1 root disk 8,  4 Aug  6 18:19 /dev/sda4
brw-r----- 1 root disk 8, 16 Aug  6 18:19 /dev/sdb
brw-r----- 1 root disk 8, 32 Aug  6 18:19 /dev/sdc
brw-r----- 1 root disk 8, 48 Aug  6 18:19 /dev/sdd
brw-r----- 1 root disk 8, 64 Aug  6 18:19 /dev/sde
```

SCSI 硬盘主设备号都是 8，而不同的盘或者不同的分区都有不同的次设备号。次设备号具体为多少并不重要，不过最大不能超过 255。USB 子系统里使用以下公式安排次设备号的，即  $\text{minor} = ((\text{dev->bus->busnum}-1) * 128) + (\text{dev->devnum}-1)$ ；而 `USB_DEVICE_MAJOR` 被定义为 189：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 tty
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
21 sg
29 fb
128 ptm
136 pts
162 raw
180 usb
189 usb_device
254 megaraid_sas_ioctl
```

189 被称为 `usb_device`，这两行代码是用于与 `usbfs` 文件系统相交互的。`dev_t` 记录下了设备的主设备号和次设备号。`dev_t` 包含两部分，主设备号部分和次设备号部分。高 12 位表示主设备号，低 20 位表示次设备号。

1395 行, `device_add()`, 设备模型中最基础的函数之一, 这个函数非常了不起。要深入追踪这个函数, 足以写一篇专题文章了。这个函数来自 `drivers/base/core.c` 中, 是设备模型里提供的函数, 从作用上来说, 这个函数一执行, 系统里就真正有了这个设备, `/sysfs` 下面也能看到了, 而且将会去遍历注册到 USB 总线上的所有的驱动程序, 如果找到合适的, 就去调用该驱动的 `probe` 函数, 对于 U 盘来说, 最终将调用 `storage_probe()` 函数, 对于 Hub 来说, 最终将调用 `hub_probe()` 函数。而传递给它们的参数, 正是我们此前获得的 `struct usb_interface` 指针和一个 `struct usb_device_id` 指针。后者正是我们在 USB 总线上寻找驱动程序的依据, 换句话说, 每个驱动程序都会把自己支持的设备定义在一张表里, 表中的每一项就是一个 `struct usb_device_id`, 然后当我们获得了一个具体设备, 我们就把该设备的实际的信息与这张表去比较, 如果找到匹配的了, 就认为该驱动支持该设备, 从而最终会调用该驱动的 `probe()` 函数。从此, 这个设备就被传递到了设备驱动。而 Hub 驱动也完成了它最重要的一项工作。

再次回到 `usb_new_device()`, 1402 行, 如果该设备不是 Root Hub, 则调用 `usb_autoresume_device()`。这个函数来自 `drivers/usb/core/driver.c`, 也是 USB Core 提供的, 是电源管理方面的函数。如果设备这时候处于 `suspended` 状态, 那么这个函数将把它唤醒, 因为我们已经要开始用它了。

最后 1406 行, 函数终于返回了。正确的话返回值为 0。

返回之后首先判断返回值, 如果不为 0 说明出错了, 那么就把 `hdev->children` 相应的那一位设置为空。要知道我们在调用 `usb_new_device` 之前可是把它设置成了 `udev` 法。

八大函数只剩下最后一个 `hub_power_remaining()`, 这个函数相对来说比较小巧玲珑。这是与电源管理相关的。虽然说刚接入的设备可能已经被设备驱动认领了, 但是作为 Hub 驱动, 自身的工作还是要处理干净的。

```

2364 static unsigned
2365 hub_power_remaining (struct usb_hub *hub)
2366 {
2367     struct usb_device *hdev = hub->hdev;
2368     int remaining;
2369     int port1;
2370
2371     if (!hub->limited_power)
2372         return 0;
2373
2374     remaining = hdev->bus_mA - hub->descriptor->bHubContrCurrent;
2375     for (port1 = 1; port1 <= hdev->maxchild; ++port1) {
2376         struct usb_device *udev = hdev->children[port1 - 1];
2377         int delta;
2378
2379         if (!udev)
2380             continue;
2381
2382         /* Unconfigured devices may not use more than 100mA,
2383          * or 8mA for OTG ports */
2384         if (udev->actconfig)

```

```

2385         delta = udev->actconfig->desc.bMaxPower * 2;
2386     else if (port1 != udev->bus->otg_port || hdev->parent)
2387         delta = 100;
2388     else
2389         delta = 8;
2390     if (delta > hub->mA_per_port)
2391         dev_warn(&udev->dev, "%dmA is over %u mA budget "
2392                 "for port %d!\n",
2393                 delta, hub->mA_per_port, port1);
2394         remaining -= delta;
2395     }
2396     if (remaining < 0) {
2397         dev_warn(hub->intfdev, "%dmA over power budget!\n",
2398                 - remaining);
2399         remaining = 0;
2400     }
2401     return remaining;
2402 }

```

`limited_power` 是当初在 `hub_configure()` 中设置的。设置了它说明能源是有限的。所以如果这个变量不为 0, 我们就要对电源精打细算。要计算出现在还能提供多大电流, 即把当前 `bus_mA` 减去 Hub 需要的电流, 以及现在连在 Hub 端口上的设备所消耗的电流, 求出剩余值来, 然后打印出来, 告诉我们还有多少电流 `budget`。bHubContrCurrent 前面在讲 `hub_configure` 时就已经说过了, 是 Hub 控制器本身最大的电流需求, 单位是 mA, 来自 Hub 描述符。

2375 行到 2395 行这段循环, 就是上面说的这个思想的具体实现。遍历每个端口进行循环。每个设备的配置描述符中 `bMaxPower` 就是该设备从 USB 总线上消耗的最大的电流, 其单位是 2mA, 所以这里要乘以 2。没有配置过的设备是不可能获得超过 100mA 电流的。

如果 `delta` 比 Hub 为每个端口提供的平均电流要大, 至少要警告一下。

然后循环完了, `remaining` 就是如其字面意义一样, 还剩下多少电流可供新的设备再接进来。从软件的角度来说, 我们是不会强行对设备采取什么措施, 最多是打印出调试信息, 进行警告。而设备如果真的遇到了供电问题, 它自然会出现异常, 它也许不能工作, 这些当然由具体的设备驱动程序去关注。

终于我们讲完了这八个函数, 可以说到这里为止, 我们已经知道了 Hub 驱动是在端口连接有变化时如何工作的, 并且更重要的是我们知道了 Hub 驱动是如何为子设备驱动服务的。回到 `hub_port_connect_change` 之后, 一切正常的话我们将会从 2579 行返回。

剩下的一些行就是错误处理代码。我们就不必再看了。因此我们将返回到 `hub_events()` 中来。不过你千万别以为看到这里你就完全明白 Hub 驱动程序了。

## 23. 是月亮惹的祸还是 spec 的错

让我们用代码来说话。2777 行，把 `hub->event_bits` 给清掉，然后读一次 Hub 的状态。`HUB_STATUS_LOCAL_POWER` 我们以前在 `hub_configure` 中见过，用来标志这个 Hub 是有专门的外接电源的还是从 USB 总线上获取电源，而 `C_HUB_LOACL_POWER` 用来标志这一位有变化。在这种情况下，先把 `C_HUB_LOCAL_POWER` 清掉，同时判断，如果是原来没有电源现在有了电源，那么可以取消 `limited_power` 了，把它设置为 0；如果原来是有电源的，而现在没了，那么没什么说的，把 `limited_power` 设置为 1。

真理和错误有时候只有一步之遥。我们来对照一下 spec 的 Table 11-19 就可以发现，代码和我的解释刚好相反。代码的意思是 `HUB_STATUS_LOCAL_POWER` 为 1，就设置 `limited_power` 为 0，反之则设置 `limited_power` 为 1。

众所周知，一个 Hub 可以用两种供电方式，一种是自带电源，即 Hub 上面自己有一根电源线，插到插座上，就行了。另一种是没有自带电源，由总线来供电。具体这个 Hub 是使用的哪种方式供电，就是从这个状态位里面可以读出来，即上面这个 Local power source，Spec 的意思是：Local Power Source 如果为 0，表明本地供电正常，即说明是自带电源；如果 Local Power Source 为 1，则说明本地供电出问题了，或者根本就没有本地供电。

而我们 Hub 设备驱动中之所以引入一个叫做 `limited_power` 的变量，就是为了记录这个现象，如果这个 Hub 有自己的电源，那么你就可以为所欲为，因为你有 Power，但是如果依靠总线来给你解决电源问题，那么驱动程序就要记录下来，因为总的资源是有限的，你占用了多少，分给别人的就少了多少，所以这种情况下设置 `limited_power` 为 1，也算是记下这么一件事。

所以说，正确的赋值应该是 `HUB_STATUS_LOCAL_POWER` 为 1，设置 `limited_power` 为 1，`HUB_STATUS_LOCAL_POWER` 为 0，设置 `limited_power` 为 0。所以这就是 Bug。有趣的是这个 Bug 自 2005 年由三剑客之一的 Alan 提出，并于 2006 年 1 月由 Greg 正式引入 Linux 内核，在连续几个稳定版的内核中隐藏了近两年，终于被我发现了。

不过我想问，这究竟是不是 spec 的错？其实 spec 算不上错，但是 spec 中这种定义是不合理的，它这一位就不该叫做 Local Power Source，因为这样一叫别人就会误以为这位为一时表示有电源，为 0 表示没有电源，所以才导致了这个 Bug，更合理的叫法应该是叫 Local Power Lost。现在好了，既然被我发现了，那么 2.6.23 的正式版内核中不会有这个 Bug 了。不过你别以为这样的 Bug 很幼稚，Alan 说过：“对一个人显然的事情未必会对另一个人显然。”很多 Bug 都存在这样的情况，当你事后去看的话，你会觉得它很不可思议，太低级了，但是有时候低级的错误你却未必能够在短时间内发现。

我们继续看代码，2791 行，对于有过流的改变也是同样的处理，因为过流可能导致端口关闭，所以重新给它上电。`hub_power_on()`，其实这个函数我们以前见过，只是当时出于情节考

虑，先忽略了，现在我们对 hub 有了这么多认识了之后再来看这个函数就好比一个大学生去看小学数学题一样简单。

```

475 static void hub_power_on(struct usb_hub *hub)
476 {
477     int port1;
478     unsigned pgood_delay = hub->descriptor->bPwrOn2PwrGood * 2;
479     u16 wHubCharacteristics =
480         le16_to_cpu(hub->descriptor->wHubCharacteristics);
481
482     /* Enable power on each port. Some hubs have reserved values
483      * of LPSM (> 2) in their descriptors, even though they are
484      * USB 2.0 hubs. Some hubs do not implement port-power switching
485      * but only emulate it. In all cases, the ports won't work
486      * unless we send these messages to the hub.
487      */
488     if ((wHubCharacteristics & HUB_CHAR_LPSM) < 2)
489         dev_dbg(hub->intfdev, "enabling power on all ports\n");
490     else
491         dev_dbg(hub->intfdev, "trying to enable port power on "
492                 "non-switchable hub\n");
493     for (port1 = 1; port1 <= hub->descriptor->bNbrPorts; port1++)
494         set_port_feature(hub->hdev, port1, USB_PORT_FEAT_POWER);
495
496     /* Wait at least 100 msec for power to become stable */
497     msleep(max(pgood_delay, (unsigned) 100));
498 }

```

关键的代码就是一行，494 行，set\_port\_feature 的 USB\_PORT\_FEAT\_POWER 这一位如果为 0，表示该端口处于 Powered-off 状态。同样，任何事情引发该端口进入 Powered-off 状态的话都会使得这一位为 0。而 set\_port\_feature 就会把这一位设置为 1，这叫做使得端口“power on”。

关于对 HUB\_CHAR\_LPSM 的判断，本来是没有必要的，关于这段代码的解释在 482 行到 487 行这段注释里面，我就不重复了，总之最终的做法就是对每个端口都执行一次 set\_port\_feature。最后 497 行，休眠，经验值是 100 ms，而 Hub 描述符里有一位 bPwrOn2PwrGood，全称就是 b-Power On to Power Good，即从打开电源到电源稳定的时间，显然我们应该在电源稳定了之后再访问每一个端口，所以这里睡眠时间就取这两个值中的较大的那一个值。

2799 行，设置 hub->activating 为 0，也就是说以上这一段被称为 activating，而这个变量本身就是一个标志而已。别忘了我们是从 hub\_actiavte 调用 kick\_khubd() 从而进入到这个 hub\_events() 的。而在 hub\_activate() 中我们设置了 hub->activating 为 1。而那个函数也是唯一一个设置这个变量为 1 的地方。

2803 行，如果是 Root Hub，并且 hub->busy\_bits[0] 为 0，hub->busy\_bits 只有在一个端口为 reset 或者 resume 时才会被设置成 1。我们暂时先不管。对于这种情况，既是 Root Hub，又没有端口处于 reset/resume 状态，调用 usb\_enable\_root\_hub\_irq() 函数，这个函数来自 drivers/usb/core/hcd.c 文件，是主机控制器驱动相关的，有些主机控制器的驱动程序提供了一个叫做 hub\_irq\_enable 的函数，这里就会去调用它，不过目前主流的 EHCI/UHCI/OHCI 都没有提

供这个函数，所以你可以认为这个函数什么也没干。这个函数的作用正如它的名字，开启端口中断。关于这个函数，涉及一些比较专业性的东西，有两种中断的方式，边缘触发和电平触发。只有电平触发的中断才需要这个函数，边缘触发的中断不需要这个函数。

2808 行进行判断，如果 Hub 的 `event_list` 没有东西了，那么就调用 `usb_autopm_enable()`，调用了这个函数这个 Hub 就可以被挂起了，强调一下，可以做某事不等于马上就做某事了，真的被挂起是有条件的，首先它的子设备必须先挂起了，而且确实一段时间内没有总线活动了，才会被挂起。

最后，释放 `hdev` 的锁，减少 `intf` 的引用计数，至此，`hub_events()` 这个函数就算结束了！对于大部分人来说，需要学习的 Hub 驱动就算学完了。因为你已经完全明白了 Hub 驱动在设备插入之后会做一些什么事情，会如何为设备服务，并最终把控制权交给设备驱动。而 `hub_thread()/hub_events()` 将永远这么循环下去。

不过我的故事可没有结束，最起码还有一个重要的函数没有讲，那就是 `hub_irq`。之前我们这里讲的内容都是基于一个事实就是我们主动去读了 Hub 端口的状态，而以后正常工作的 Hub 驱动是不会莫名其妙就去读 Hub 端口状态，只有发生了中断才会去读。而这个中断的服务函数就是 `hub_irq()`，也就是说，凡是真正的有端口变化事件发生，`hub_irq` 就会被调用，而 `hub_irq()` 最终会调用 `kick_khubd()`，触发 Hub 的 `event_list`，于是再次调用 `hub_events()` 函数。

## 24. 所谓的热插拔

我们曾经在 `hub_configure` 中讲过中断传输，当时调用了 `usb_fill_int_urb()` 函数，并且把 `hub_irq` 作为一个参数传递了进去，最终把 `urb->complete` 赋值为 `hub_irq`。然后，主机控制器会定期询问 Hub，每当 Hub 端口上有一个设备插入或者拔除时，它就会向主机控制器打小报告。具体来说，从硬件的角度看，就是 Hub 会向主机控制器返回一些信息，或者说 Data，这个 Data 被称作“Hub and Port Status Change Bitmap”，而从软件角度来看，主机控制器的驱动程序接下来会在处理好这个过程的 `urb` 之后，调用该 `urb` 的 `complete` 函数，对于 Hub 来说，这个函数就是 `hub_irq()`。

```
338 static void hub_irq(struct urb *urb)
339 {
340     struct usb_hub *hub = urb->context;
341     int status;
342     int i;
343     unsigned long bits;
344
345     switch (urb->status) {
346     case -ENOENT:          /* synchronous unlink */
347     case -ECONNRESET:     /* async unlink */
```

```

348     case -ESHUTDOWN:          /* hardware going away */
349         return;
350
351     default:                    /* presumably an error */
352         /* Cause a hub reset after 10 consecutive errors */
353         dev_dbg (hub->intfdev, "transfer --> %d\n", urb->status);
354         if ((++hub->nerrors < 10) || hub->error)
355             goto resubmit;
356         hub->error = urb->status;
357         /* FALL THROUGH */
358
359     /* let khubd handle things */
360     case 0:                      /* we got data: port status changed */
361         bits = 0;
362         for (i = 0; i < urb->actual_length; ++i)
363             bits |= ((unsigned long) ((*hub->buffer)[i]))
364                     << (i*8);
365         hub->event_bits[0] = bits;
366         break;
367     }
368
369     hub->nerrors = 0;
370
371     /* Something happened, let khubd figure it out */
372     kick_khubd(hub);
373
374     resubmit:
375     if (hub->quiescing)
376         return;
377
378     if ((status = usb_submit_urb (hub->urb, GFP_ATOMIC)) != 0
379         && status != -ENODEV && status != -EPERM)
380         dev_err (hub->intfdev, "resubmit --> %d\n", status);
381 }

```

你问这个参数 `urb` 是哪个 `urb`?告诉你, 中断传输就是只有一个 `urb`, 不是说像 `bulk` 传输那样每次开启一次传输都要有申请一个 `urb`, 提交 `urb`。对于中断传输, 一个 `urb` 就可以了, 反复利用, 所以我们只有一次调用 `usb_fill_int_urb()` 函数。这正体现了中断交互的周期性。

340 行, 当初我们填充 `urb` 时, `urb->context` 就是赋的 `hub`, 所以现在这句话就可以获得那个 `Hub`。

345 行开始判断 `urb` 的状态, 前三种都是出错了, 直接返回。

351 的 `default` 和 `case 0`。这段代码是我认为最有技术含量的一段代码。我以为这里 `default` 不管 `case` 等于 0 与否都会执行, 结果半天没看懂, 后来我明白了, 其实当 `urb->status` 为 0 时, `default` 那一段是不会执行的。

所以这段代码就很好理解了。一开始 `hub->error` 为 0, `hub->nerrors` 也为 0, 所以 `default` 这一段很明显, `goto resubmit`, `resubmit` 那一段就是重新调用了一次 `usb_submit_urb()` 而已。当然, 还判断了 `hub->quiescing`。这个变量初始值为 1, 但是我们前面在 `hub_activate` 里把它设置为了 0, 有一个函数会把它设置为 1, 这个函数就是 `hub_quiesce()`, 而调用后者的只有两个函数, 一

一个是 `hub_suspend()`，一个是 `hub_pre_reset()`。于是，这里的意思就很明确了，如果 Hub 被挂起了，或者要被 `reset` 了，那么就不用重新提交 `urb` 了，`hub_irq()` 函数直接返回吧。

再看 case 0，`urb->status` 为 0，说明这个 `urb` 被顺利地处理了，主机控制器获得了想要的数  
据，即“Hub and Port Status Change Bitmap”，因为我们当初调用 `usb_fill_int_urb` 时，把  
`*hub->buffer` 传递给了 `urb->transfer_buffer`，所以这个数据现在就在 `hub->buffer` 中。至于这个  
bitmap 是什么样子，我们查看 spec 中的 Figure 11-22，首先这幅图为我们解答了一个很大的疑  
惑。当时在 `hub_events()` 中您大概还有一些地方不是很清楚。现在我们可以来弄清楚它们了。

我们回过头来看，`struct usb_hub` 中，`unsigned long event_bits[1]`，首先这是一个数组，其次  
这个数组只有一个元素，而这一个元素恰恰就是对应这里的 Bitmap，即所谓的位图，每一位都  
有其作用。一个 `unsigned long` 至少 4 个 Bytes，即 32 个 bits。所以够用了，而我们看到这张图  
中，`bit0` 和其他 `bit` 是不一样的，`bit 0` 表示 Hub 有变化，而其他 `bit` 则具体表示某一个端口有  
没有变化，即 `bit 1` 表示端口 1 有变化，`bit 2` 表示端口 2 有变化，如果一个端口没有变化，对应的  
那一位就是 0。

所以我们可以回到 `hub_events()` 函数中来，查看当时我们是如何判断 `hub->event_bits` 的，  
当时我们有这么一小段代码：

```
2685      /* deal with port status changes */
2686      for (i = 1; i <= hub->descriptor->bNbrPorts; i++) {
2687          if (test_bit(i, hub->busy_bits))
2688              continue;
2689          connect_change = test_bit(i, hub->change_bits);
2690          if (!test_and_clear_bit(i, hub->event_bits) &&
2691              !connect_change && !hub->activating)
2692              continue;
```

看到了吗？循环指数 `i` 从 1 开始，有多少端口就循环多少次，而对 `event_bits` 的测试，即 2690  
判断的是 bitmap 中 `bit 1`，`bit 2`，...，`bit N`，而不需要判断 `bit 0`。

反过来，如果具体每个端口没有变化，而变化的是 Hub 的整体，比如，Local Power 有变  
化，比如 Overcurrent 有变化，我们则需要判断的是 `bit 0`。即当时我们在 `hub_events()` 中看到的  
下面这段代码。

```
2776      /* deal with hub status changes */
2777      if (test_and_clear_bit(0, hub->event_bits) == 0)
2778          ; /* do nothing */
2779      else if (hub_hub_status(hub, &hubstatus, &hubchange) < 0)
2780          dev_err (hub_dev, "get_hub_status failed\n");
2781      else {
2782          if (hubchange & HUB_CHANGE_LOCAL_POWER) {
2783              dev_dbg (hub_dev, "power change\n");
2784              clear_hub_feature(hdev, C_HUB_LOCAL_POWER);
2785              if (hubstatus & HUB_STATUS_LOCAL_POWER)
2786                  /* FIXME: Is this always true? */
2787                  hub->limited_power = 0;
2788          } else
```



```

2789             hub->limited_power = 1;
2790         }
2791         if (hubchange & HUB_CHANGE_OVERCURRENT) {
2792             dev_dbg (hub_dev, "overcurrent change\n");
2793             msleep(500);    /* Cool down */
2794             clear_hub_feature(hdev, C_HUB_OVER_CURRENT);
2795             hub_power_on(hub);
2796         }
2797     }

```

而这样我们就很清楚 `hub_events()` 的整体思路了，判断每个端口是否有变化，如果有变化就去处理它，没有变化也没有关系，接下来判断是否 Hub 整体上有变化，如果有变化，那么也去处理它。

关于这个 `actual_length`，我就不啰嗦了。因为每一个 Hub 的端口是不一样的，所以这张 bitmap 的长度就不一样，比如说你是 16 个端口，那么这个 bitmap 最多就只要 16+1 个 bit 就足够了。而 `actual_length` 就是 3，即 3 个 Bytes。因为 3 个 Bytes 等于 24 个 bits，足以容纳 16+1 个 bits 了。而 `struct usb_hub` 中，`buffer` 是这样一个成员，`char (*buffer)[8]`，所以 3 个 Bytes 就意味着这个 `buffer` 的前三个元素里承载着我们的希望。这样我们就不难理解这里这个 `hub->event_bits` 是如何变成这张 bitmap 的了。

369 行，把 `nerrors` 清零。

最关键的当然还是 372 行，再次调用 `kick_khubd()` 函数，于是会再一次触发 `hub_events()`。

而 `hub_irq()` 函数也就到这里了。这个函数不长，可是很重要，其中最重要的正是最后这句 `kick_khubd()`。而这也就是所谓的热插拔的实现路径，要知道我们上次分析 `kick_khubd` 时是在 Hub 初始化时，即那次针对的情况是设备一开始就插在了 Hub 上，而这里再次调用 `kick_khubd` 才是真正的在使用过程中。

# 第 3 篇

## Linux 那些事儿之我是 UHCI

1. 引子 .....	256	12. 一个函数引发的故事（四） .....	309
2. 开户和销户 .....	258	13. 一个函数引发的故事（五） .....	311
3. PCI，我们来了！ .....	262	14. 寂寞在唱歌 .....	313
4. I/O 内存和 I/O 端口 .....	270	15. Root Hub 的控制传输（一） .....	321
5. 传说中的 DMA .....	275	16. Root Hub 的控制传输（二） .....	327
6. 来来，我是一条总线， 线线线线线 .....	281	17. 非 Root Hub 的批量传输 .....	339
7. 主机控制器的初始化 .....	285	18. 传说中的中断服务程序（ISR） .....	345
8. 有一种资源，叫中断 .....	293	19. Root Hub 的中断传输 .....	362
9. 一个函数引发的故事（一） .....	295	20. 非 Root Hub 的中断传输 .....	364
10. 一个函数引发的故事（二） .....	298	21. 等时传输 .....	375
11. 一个函数引发的故事（三） .....	303	22. “脱”就一个字 .....	381

## 1. 引子

UHCI, Universal Host Controller Interface, 是一种 USB 主机控制器的接口规范。它是 Intel 公司提出来的,“江湖”中把遵守它的硬件称为 UHCI 主机控制器。在 Linux 中,这种硬件叫做 HC,或者说 Host Controller,而把与它对应的软件叫做 HCD,即 HC Driver。Linux 中的 HCD 所对应的模块叫做 uhci-hcd。

当我们看一个模块时,首先是看 Kconfig 和 Makefile 文件。在 drivers/usb/host/Kconfig 文件中:

```

161 config USB_UHCI_HCD
162     tristate "UHCI HCD (most Intel and VIA) support"
163     depends on USB && PCI
164     ---help---
165     The Universal Host Controller Interface is a standard by Intel for
166     accessing the USB hardware in the PC (which is also called the USB
167     host controller). If your USB host controller conforms to this
168     standard, you may want to say Y, but see below. All recent boards
169     with Intel PCI chipsets (like intel 430TX, 440FX, 440LX, 440BX,
170     i810, i820) conform to this standard. Also all VIA PCI chipsets
171     (like VIA VP2, VP3, MVP3, Apollo Pro, Apollo Pro II or Apollo Pro
172     133). If unsure, say Y.
173
174     To compile this driver as a module, choose M here: the
175     module will be called uhci-hcd.

```

请注意第 163 句 `depends on USB && PCI`, 意即这个选项依赖于另外两个选项: `CONFIG_USB` 和 `CONFIG_PCI`。很显然这两个选项分别代表着 Linux 中 USB 和 PCI 的核心代码。

UHCI 作为 USB 主机控制器的接口,依赖于 USB 核心,很正常,但为何它也依赖于 PCI 核心代码呢?理由很简单,UHCI 主机控制器本身通常是 PCI 设备,即通常它会插在 PCI 插槽里,或者直接就集成在主板上。但总之,大多数 UHCI 主机控制器是连在 PCI 总线上的。所以,写 UHCI 驱动程序就不得不了解一点 PCI 设备驱动程序。

先用 `lspci` 命令看一下:

```

localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # lspci | grep USB
00:1d.0 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #1 (rev
09)
00:1d.1 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #2 (rev
09)
00:1d.2 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #3 (rev
09)

```

```
00:1d.7 USB Controller: Intel Corporation Enterprise Southbridge EHCI USB (rev 09)
```

比如在我的计算机里，就有三个 UHCI 主机控制器，以及另一个主机控制器和 EHCI 主机控制器。它们都是 PCI 设备。

接着来看 Makefile:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # cat Makefile
#
# Makefile for USB Host Controller Drivers
#

ifeq ($(CONFIG_USB_DEBUG),y)
    EXTRA_CFLAGS      += -DDEBUG
endif

obj-$(CONFIG_PCI)      += pci-quirks.o

obj-$(CONFIG_USB_EHCI_HCD)      += ehci-hcd.o
obj-$(CONFIG_USB_ISP116X_HCD)  += isp116x-hcd.o
obj-$(CONFIG_USB_OHCI_HCD)     += ohci-hcd.o
obj-$(CONFIG_USB_UHCI_HCD)     += uhci-hcd.o
obj-$(CONFIG_USB_SL811_HCD)    += sl811-hcd.o
obj-$(CONFIG_USB_SL811_CS)     += sl811_cs.o
obj-$(CONFIG_USB_U132_HCD)    += u132-hcd.o
```

很显然，我们要的就是与 CONFIG\_USB\_UHCI\_HCD 对应的 uhci-hcd.o 模块。而与 uhci-hcd.o 最相关的就是与之同名的 C 文件。这是它的源文件。在 drivers/usb/host/uhci-hcd.c 的最后 7 行，可以看到：

```
969 module_init(uhci_hcd_init);
970 module_exit(uhci_hcd_cleanup);
971
972 MODULE_AUTHOR(DRIVER_AUTHOR);
973 MODULE_DESCRIPTION(DRIVER_DESC);
974 MODULE_LICENSE("GPL");
```

正如每个女人都应该有一支口红一样，每个模块都应该有两个宏：module\_init 和 module\_exit，分别用来初始化和注销自己。而这两行代码的意思就是说 uhci\_hcd\_init 函数将会在加载这个模块时被调用，uhci\_hcd\_cleanup 则将在卸载这个模块时被执行。

所以，只能从 uhci\_hcd\_init 开始我们的故事：

```
917 static int __init uhci_hcd_init(void)
918 {
919     int retval = -ENOMEM;
920
921     printk(KERN_INFO DRIVER_DESC " " DRIVER_VERSION "%s\n",
922            ignore_oc ? ", overcurrent ignored" : "");
923
924     if (usb_disabled())
925         return -ENODEV;
926
```

```

927     if (DEBUG_CONFIGURED) {
928         errbuf = kmalloc(ERRBUF_LEN, GFP_KERNEL);
929         if (!errbuf)
930             goto errbuf_failed;
931         uhci_debugfs_root = debugfs_create_dir("uhci", NULL);
932         if (!uhci_debugfs_root)
933             goto debug_failed;
934     }
935
936     uhci_up_cacheop = kmem_cache_create("uhci_urb_priv",
937         sizeof(struct urb_priv), 0, 0, NULL, NULL);
938     if (!uhci_up_cacheop)
939         goto up_failed;
940
941     retval = pci_register_driver(&uhci_pci_driver);
942     if (retval)
943         goto init_failed;
944
945     return 0;
946
947 init_failed:
948     kmem_cache_destroy(uhci_up_cacheop);
949
950 up_failed:
951     debugfs_remove(uhci_debugfs_root);
952
953 debug_failed:
954     kfree(errbuf);
955
956 errbuf_failed:
957
958     return retval;
959 }

```

## 2. 开户和销户

之所以说 `uhci_hcd_init` 有技术含量，并不是说它包含多么精巧的算法，包含多么复杂的数据结构，而是因为这其中涉及了很多东西。首先 924 行，`usb_disable` 涉及了 Linux 中的内核参数的概念，928 行的 `kmalloc` 和 936 行的 `kmem_cache_create` 涉及了 Linux 内核中内存申请的问题，931 行 `debugfs_create_dir` 则涉及了一个虚拟的文件系统 `debugfs`，而 941 行 `pci_register_driver` 则涉及 Linux 中 PCI 设备驱动程序的注册。

这么多东西往这里一堆，其复杂程度立马就上来了。

### 内核参数

什么是内核参数？看一下 `grub` 文件：

```
title SUSE Linux Enterprise Server 10 (kdb enabled)
```

```
kernel (hd0,2)/boot/vmlinuz-2.6.22.1-test root=/dev/hda3 resume=/dev/hda2
splash=silent showopts
initrd /boot/initrd-2.6.22.1-test
```

kernel 那行都是内核参数，比如 root， resume， splash， showopts。其中“root=”代表的是 root 文件系统的位置，“resume=”代表的是用于 software suspend 恢复的分区。而 USB 子系统也准备了 nousb 参数。所以如果往这一行后面加上 nousb，则意味着系统不需要支持 USB，即把 USB 子系统给“disable”掉了。换句话说，usb\_disabled 返回的就是 nousb 的值。在 drivers/usb/core/usb.c 中也能看到这个函数：

```
852 /*
853  * for external read access to <nousb>
854  */
855 int usb_disabled(void)
856 {
857     return nousb;
858 }
```

## 申请内存

用来申请内存的两个函数分别是 kmalloc 和 kmem\_cache\_create。应该很熟悉 kmalloc。而 kmem\_cache\_create 则是传说中的 slab “现身”了。传统上，kmem\_cache\_create 是 slab 分配器的接口函数，用于创建一个“内存池”cache 创建了一个 cache 之后，就可以用另一个函数 kmem\_cache\_zalloc 来申请内存，使用 kmem\_cache\_free 来释放内存。可以使用 kmem\_cache\_destroy 来彻底释放这个内存池。

这里重点是每次用 kmem\_cache\_zalloc 申请内存的大小是一样的，即在 kmem\_cache\_create 中的第二个参数所指定的，比如 sizeof(struct urb\_priv)，即以后用 kmem\_cache\_zalloc 申请的内存总是这么大。而这里 kmem\_cache\_create 的返回值就是创建好的那个 cache。而且这个返回值被赋给了 uhci\_up\_cachep。它是一个 struct kmem\_cache 的结构体指针。所以以后用 kmem\_cache\_zalloc 时只要把 uhci\_up\_cachep 作为参数即可得到想要的内存。对于 kmem\_cache\_free 和 kmem\_cache\_destroy 也一样。

理解这些函数最简单的比喻就是，去沃尔玛超市购物，超市里给你提供了篮子，你可以把你需要的东西装在篮子里，但大家都知道超市里的篮子数量是有限的，虽然存在超市篮子不够的情况，但是沃尔玛在开张之前肯定会准备了足够多的篮子，比如它订做了一个仓库的篮子，每个篮子都一样大。即一开始沃尔玛方就调用了 kmem\_cache\_create 做了一池的篮子，而你每次去就是使用 kmem\_cache\_zalloc 去拿一个篮子即可，而当你付款之后要离开了，你又调用 kmem\_cache\_free 去归还篮子。如果每个人都这样做的话，你下次去了要用篮子又可以用 kmem\_cache\_zalloc 再拿一个。

而一旦哪天沃尔玛宣武门分店连续亏损，店子倒闭，它就可以调用 kmem\_cache\_destroy 把篮子全都毁掉，当然更形象的例子是它把篮子转移到别的店去，比如知春路分店。那么从整个沃尔玛公司来看，可以供来装东西的容器总容积还是没有变，正如你的计算机总的内存是不会

变的。假如公司将来又打算在国贸开一家分店，那么它可以再次调用 `kmem_cache_create`。而对你来说，你并不需要知道一池内存到底有多少，就像你永远不用知道沃尔玛知春路店究竟有多少个篮子一样。

## 调试信息

好了，第三，`debugfs_create_dir`，传说中的 `debugfs` 也“现身了”。很多事情都是早已注定的，原本以为写设备驱动的只要懂一些硬件规范就可以了，后来终于在眼泪中明白，有些人一旦写代码就会越写越复杂。如今的内核代码早已不那么单纯了。

以前我一直以为，Linux 中 PCI 子系统和 USB 子系统的掌门人 Greg 同志只是一个花拳绣腿的家伙，只是每天忙着到处演讲、开会，而不干正经事。后来我发现，其实不然，Greg 其实还是干了很多有意义的事情，不得不承认，Greg 是条汉子！`debugfs` 就是他开发的一个虚拟的文件系统，专门用于输出调试信息。这个文件系统默认是被挂载在 `/sys/kernel/debug` 下的，比如：

```
localhost:~ # mount
/dev/hda3 on / type reiserfs (rw,acl,user_xattr)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
debugfs on /sys/kernel/debug type debugfs (rw)
udev on /dev type tmpfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
```

这个文件系统是专门为开发人员准备的，在配置内核时可以编译也可以不编译进去。其对应的 `Kconfig` 文件是 `lib/Kconfig.debug`：

```
50 config DEBUG_FS
51     bool "Debug Filesystem"
52     depends on SYSFS
53     help
54         debugfs is a virtual file system that kernel developers use to put
55         debugging files into. Enable this option to be able to read and
56         write to these files.
57
58     If unsure, say N.
```

很有意思的是，这个文件系统居然依赖于另一个文件系统——`sysfs`。如果用 `make menuconfig` 命令编译内核，则在 `Kernel hacking` 下面找到它，即 `DEBUG_FS`。我们不去深入研究这个文件系统，但是对于它的接口函数是有必要了解一下的。

首先，`debugfs_create_dir` 就是创建一个目录。像这里这么一行的作用就是在 `/sys/kernel/debug` 下面创建一个叫做 `uhci` 的目录，比如加载了 `uhci-hcd` 这个模块，就能看到 `uhci` 目录：

```
localhost:/usr/src/linux-2.6.22.1 # ls /sys/kernel/debug/
kprobes uhci
```

这个函数的返回值是文件系统里最经典的一个 `struct dentry` 结构体指针。而这里我们把返回值赋给了 `struct dentry` 指针 `uhci_debugfs_root`。它被定义在 `drivers/usb/host/uhci-debug.c` 中：

```
20 static struct dentry *uhci_debugfs_root;
```

显然这个指针对 uhci-hcd 模块来说是到处可以引用的。且可用 `debugfs_create_file` 函数在 uhci 目录下创建文件。这在 `uhci_start` 函数中也会介绍。而以后删除这个目录的任务就在 `uhci_hcd_cleanup` 中，它只要调用 `debugfs_remove` 函数即可。

## 注册 PCI

现在剩下第四个问题，`pci_register_driver`，其实一路走来应该多少有感觉，虽然没见过这个函数，但是能感觉出它和 `usb_register_driver` 注册 USB 驱动作用相同，注册 PCI 驱动。请注意参数 `uhci_pci_driver`。

```
894 static const struct pci_device_id uhci_pci_ids[] = { {
895     /* handle any USB UHCI controller */
896     PCI_DEVICE_CLASS(PCI_CLASS_SERIAL_USB_UHCI, ~0),
897     .driver_data = (unsigned long) &uhci_driver,
898 }, { /* end: all zeroes */ }
899 };
900
901 MODULE_DEVICE_TABLE(pci, uhci_pci_ids);
902
903 static struct pci_driver uhci_pci_driver = {
904     .name = (char *)hcd_name,
905     .id_table = uhci_pci_ids,
906
907     .probe = usb_hcd_pci_probe,
908     .remove = usb_hcd_pci_remove,
909     .shutdown = uhci_shutdown,
910
911 #ifdef CONFIG_PM
912     .suspend = usb_hcd_pci_suspend,
913     .resume = usb_hcd_pci_resume,
914 #endif /* PM */
915 };
```

这里 `PCI_CLASS_SERIAL_USB_UHCI` 是 `0x0c0300`，03 表示类别为 03，代表 USB；而 00 代表 UHCI。OHCI 是 `0x0c0310`，而 EHCI 则是 `0x0c0320`。而 PCI spec 规定：最前面两位的 0c 代表所有串行总线控制器。

所以不难知道，真正的故事将从哪个函数开始：probe 函数，即 `usb_hcd_pci_probe`。在讲这个函数之前在本节的最后把 `uhci_hcd_cleanup` 也给贴出来：

```
961 static void __exit uhci_hcd_cleanup(void)
962 {
963     pci_unregister_driver(&uhci_pci_driver);
964     kmem_cache_destroy(uhci_up_cache);
965     debugfs_remove(uhci_debugfs_root);
966     kfree(errbuf);
967 }
```

总之，一个模块就是这样，写一个注册函数，再写一个注销函数即可。Linux 中模块机制的便利就是：当想用它时可以用 `modprobe` 或者 `insmod` 加载它；当不想用它时，可以用 `rmmod`



卸载它。加载就是注册，卸载就是注销，就像银行里的开户和销户，你可以随意享受这种服务。

### 3. PCI，我们来了！

神圣的 PCI 设备驱动程序——`usb_hcd_pci_probe` 即将带领我们开启新的篇章，开始 PCI 世界之旅，也将开始一段全新的体验。

细心的你或许注意到了，关于 HCD 的代码，被分布于两个目录：`drivers/usb/core/`和 `drivers/usb/host/`，其中前者包含三个相关的文件：`hcd-pci.c`，`hcd.c` 和 `hcd.h`，这是一些公共的代码。因为 USB 主机控制器有很多种，光从接口来说，目前就有 UHCI，OHCI 和 EHCI，谁知道以后还会有更多呢。而这些主机控制器的驱动程序有一些代码是相同的，所以就把它提取出来，专门写在某几个文件中，因此有了这种格局。光就某一种具体的 HCD 代码，还是在 `drivers/usb/host/`下面，比如与 UHCI 相关的代码就是以下几个文件：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # ls uhci-*
uhci-debug.c uhci-hcd.c uhci-hcd.h uhci-hub.c uhci-q.c
```

就 UHCI 的驱动来说，其四大函数指针 `probe/remove/suspend/resume` 都是指向一些公共函数，都定义于 `drivers/usb/core/hcd-pci.c` 中。只有一个 `shutdown` 指针指向的函数 `uhci_shutdown` 是它自己定义的来自 `drivers/usb/host/uhci-hcd.c` 中。

`probe` 函数，即 `usb_hcd_pci_probe` 来自 `drivers/usb/core/hcd-pci.c`：

```
58 int usb_hcd_pci_probe (struct pci_dev *dev, const struct pci_device_id *id)
59 {
60     struct hc_driver      *driver;
61     struct usb_hcd        *hcd;
62     int                    retval;
63
64     if (usb_disabled())
65         return -ENODEV;
66
67     if (!id || !(driver = (struct hc_driver *) id->driver_data))
68         return -EINVAL;
69
70     if (pci_enable_device (dev) < 0)
71         return -ENODEV;
72     dev->current_state = PCI_D0;
73     dev->power.power_state = PMSG_ON;
74
75     if (!dev->irq) {
76         dev_err (&dev->dev,
77                 "Found HC with no IRQ. Check BIOS/PCI %s setup!\n",
78                 pci_name(dev));
79         retval = -ENODEV;
80         goto err1;
```

```

81     }
82
83     hcd = usb_create_hcd (driver, &dev->dev, pci_name(dev));
84     if (!hcd) {
85         retval = -ENOMEM;
86         goto err1;
87     }
88
89     if (driver->flags & HCD_MEMORY) {          // EHCI, OHCI
90         hcd->rsrc_start = pci_resource_start (dev, 0);
91         hcd->rsrc_len = pci_resource_len (dev, 0);
92         if (!request_mem_region (hcd->rsrc_start, hcd->rsrc_len,
93                                 driver->description)) {
94             dev_dbg (&dev->dev, "controller already in use\n");
95             retval = -EBUSY;
96             goto err2;
97         }
98         hcd->regs = ioremap_nocache (hcd->rsrc_start, hcd->rsrc_len);
99         if (hcd->regs == NULL) {
100             dev_dbg (&dev->dev, "error mapping memory\n");
101             retval = -EFAULT;
102             goto err3;
103         }
104     } else {                                  // UHCI
105         int      region;
106
107         for (region = 0; region < PCI_ROM_RESOURCE; region++) {
108             if (!(pci_resource_flags (dev, region) &
109                 IORESOURCE_IO))
110                 continue;
111
112             hcd->rsrc_start = pci_resource_start (dev, region);
113             hcd->rsrc_len = pci_resource_len (dev, region);
114             if (request_region (hcd->rsrc_start, hcd->rsrc_len,
115                               driver->description))
116                 break;
117         }
118         if (region == PCI_ROM_RESOURCE) {
119             dev_dbg (&dev->dev, "no i/o regions available\n");
120             retval = -EBUSY;
121             goto err1;
122         }
123     }
124
125     pci_set_master (dev);
126
127     retval = usb_add_hcd (hcd, dev->irq, IRQF_SHARED);
128     if (retval != 0)
129         goto err4;
130     goto err4;
131     return retval;
132
133 err4:
134     if (driver->flags & HCD_MEMORY) {
135         iounmap (hcd->regs);
136 err3:
137         release_mem_region (hcd->rsrc_start, hcd->rsrc_len);
138     } else
139         release_region (hcd->rsrc_start, hcd->rsrc_len);

```

```
140 err2:
141     usb_put_hcd (hcd);
142 err1:
143     pci_disable_device (dev);
144     dev_err (&dev->dev, "init %s fail, %d\n", pci_name(dev), retval);
145     return retval;
146 }
```

PCI 设备驱动程序肯定要比 USB 设备驱动程序复杂。其他我不说，光凭这幅经典的 PCI 标准配置寄存器的图就够我们这些新手们研究半天的（如图 3.3.1 所示）。

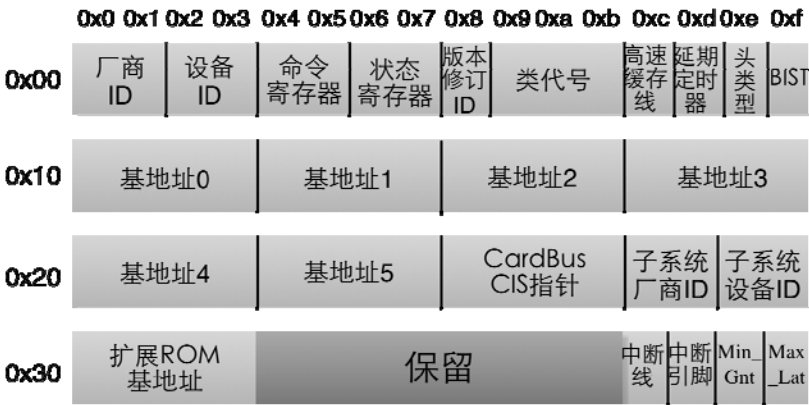


图 3.3.1 PCI 配置寄存器

看明白这张图就算对 PCI 设备驱动有一点认识，看不明白的话就说明还没入门。不过不要慌，我也不懂，让我陪着你一起结合代码来看。不过从此刻开始，这张图将被我们无数次地提起。为了便于称呼，我们给这张图取个好记的名字，就叫“清明上坟图”吧，简称“上坟图”。这张图在整个 PCI 世界里的作用就相当于我们学习化学的教材中最后几页里附上的那个《化学元素周期表》。写 PCI 设备驱动的人对于这张图的熟悉程度就要达到我们当时那种随口就能喊出“氢氦锂铍硼碳氮氧氟氖钠镁铝硅磷硫氯氩钾钙”的境界。

70 行，pci\_enable\_device()，在使用一个 PCI 设备之前，必须调用 pci\_enable\_device 激活它，该函数会调用底层代码激活 PCI 设备上的 I/O 资源和内存资源。而 143 行那个 pci\_disable\_device 则恰恰是做一些与之相反的事情。任何一个 PCI 设备驱动程序都会调用这两个函数。只有在激活了设备之后，驱动程序才可以访问它的资源。

72 行，73 行，这里的 dev 是 struct pci\_dev 结构体指针，它有一个成员：pci\_power\_t current\_state，用来记录该设备的当前电源状态，这个世界上除了我们熟知的 PCI spec 以外，还有一个规范叫做 PCI Power Management spec，它专门为 PCI 设备定义那些电源管理方面的接口。按这个规范，PCI 设备一共可以有四种电源状态：D0，D1，D2，D3。正常的工作状态就是 D0，而 D3 是耗电最少的状态，也就意味着设备“Power off”了。在 include/linux/pci.h 中有关于这些状态的定义：

```

74 #define PCI_D0          ((pci_power_t __force) 0)
75 #define PCI_D1          ((pci_power_t __force) 1)
76 #define PCI_D2          ((pci_power_t __force) 2)
77 #define PCI_D3hot       ((pci_power_t __force) 3)
78 #define PCI_D3cold      ((pci_power_t __force) 4)
79 #define PCI_UNKNOWN     ((pci_power_t __force) 5)
80 #define PCI_POWER_ERROR ((pci_power_t __force) -1)

```

继续看 `usb_hcd_pci_probe()` 的 75 行, `dev->irq`, `struct pci_dev` 有这么一个成员: `unsigned int irq`。这个意思很明显, 中断号, 它来自哪里? 好, 让我们第一次说一下这张“清明上坟图”了, 每一个 PCI 设备都有一堆寄存器, 厂商就是按着这张图来设计自己的设备。这张图里全都是寄存器, 但是并非所有设备都拥有全部寄存器, 其中有些是必选的, 有些是可选的, 就好比我们大学里面的必修课和选修课。比如, 厂商 ID、设备 ID 和类代号这就是必选的, 它们就用来标志一个设备, 而很多厂商也是会利用子系统厂商 ID 和子系统设备 ID 的, 因为可以进一步地细分设备。

仔细数一数, 这张图里一共是 64 个字节。而其中倒数第 4 个字节, 即 Byte 60, 记录的正是该设备可以使用的中断号。在系统初始化时这个值就已经被写进去了, 所以对于写设备驱动的人来说, 不需要考虑太多。这就是 `dev->irq` 这行的意思。USB 主机控制器必须有中断号, 否则没法正常工作。

接下来, `usb_create_hcd()`, 才正式进入 HCD 的概念。这个函数来自 `drivers/usb/core/hcd.c`:

```

1493 struct usb_hcd *usb_create_hcd (const struct hc_driver *driver,
1494                                struct device *dev, char *bus_name)
1495 {
1496     struct usb_hcd *hcd;
1497
1498     hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, FP_KERNEL);
1499     if (!hcd) {
1500         dev_dbg (dev, "hcd alloc failed\n");
1501         return NULL;
1502     }
1503     dev_set_drvdata(dev, hcd);
1504     kref_init(&hcd->kref);
1505
1506     usb_bus_init(&hcd->self);
1507     hcd->self.controller = dev;
1508     hcd->self.bus_name = bus_name;
1509     hcd->self.uses_dma = (dev->dma_mask != NULL);
1510
1511     init_timer(&hcd->rh_timer);
1512     hcd->rh_timer.function = rh_timer_func;
1513     hcd->rh_timer.data = (unsigned long) hcd;
1514 #ifdef CONFIG_PM
1515     INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
1516 #endif
1517
1518     hcd->driver = driver;
1519     hcd->product_desc = (driver->product_desc) ? driver->product_desc :
1520         "USB Host Controller";
1521

```

```

1522     return hcd;
1523 }

```

usb\_create\_hcd()的第一个参数 struct hc\_driver，这个结构体掀开了我们对 USB 主机控制器驱动的认识，它来自 drivers/usb/core/hcd.h:

```

149 struct hc_driver {
150     const char      *description; /* "ehci-hcd" etc */
151     const char      *product_desc; /* product/vendor string */
152     size_t          hcd_priv_size; /* size of private data */
153
154     /* irq handler */
155     irqreturn_t      (*irq) (struct usb_hcd *hcd);
156
157     int              flags;
158 #define HCD_MEMORY      0x0001 /* HC regs use memory (else I/O) */
159 #define HCD_USB11      0x0010 /* USB 1.1 */
160 #define HCD_USB2      0x0020 /* USB 2.0 */
161
162     /* called to init HCD and root hub */
163     int              (*reset) (struct usb_hcd *hcd);
164     int              (*start) (struct usb_hcd *hcd);
165
166     /* NOTE: these suspend/resume calls relate to the HC as
167      * a whole, not just the root hub; they're for PCI bus glue.
168      */
169     /* called after suspending the hub, before entering D3 etc */
170     int              (*suspend) (struct usb_hcd *hcd, pm_message_t message);
171
172     /* called after entering D0 (etc), before resuming the hub */
173     int              (*resume) (struct usb_hcd *hcd);
174
175     /* cleanly make HCD stop writing memory and doing I/O */
176     void             (*stop) (struct usb_hcd *hcd);
177
178     /* shutdown HCD */
179     void             (*shutdown) (struct usb_hcd *hcd);
180
181     /* return current frame number */
182     int              (*get_frame_number) (struct usb_hcd *hcd);
183
184     /* manage i/o requests, device state */
185     int              (*urb_enqueue) (struct usb_hcd *hcd,
186                                     struct usb_host_endpoint *ep,
187                                     struct urb *urb,
188                                     gfp_t mem_flags);
189     int              (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
190
191     /* hw synch, freeing endpoint resources that urb_dequeue can't */
192     void             (*endpoint_disable) (struct usb_hcd *hcd,
193                                           struct usb_host_endpoint *ep);
194
195     /* root hub support */
196     int              (*hub_status_data) (struct usb_hcd *hcd, char *buf);
197     int              (*hub_control) (struct usb_hcd *hcd,
198                                     u16 typeReq, u16 wValue, u16 wIndex,
199                                     char *buf, u16 wLength);
200     int              (*bus_suspend) (struct usb_hcd *);

```

```

201     int             (*bus_resume)(struct usb_hcd *);
202     int             (*start_port_reset)(struct usb_hcd *, unsigned port_num);
203     void             (*hub_irq_enable)(struct usb_hcd *);
204     /* Needed only if port-change IRQs are level-triggered */
205 };

```

说句良心话，你说这么长的一个结构体，要我怎么看？写代码的就不能写点良心代码？反正吧，每个 HCD 都得对应这么一个结构体变量。比如 UHCI，在 `drivers/usb/host/uhci-hcd` 中就有这么一段：

```

862 static const char hcd_name[] = "uhci_hcd";
863
864 static const struct hc_driver uhci_driver = {
865     .description =      hcd_name,
866     .product_desc =     "UHCI Host Controller",
867     .hcd_priv_size =    sizeof(struct uhci_hcd),
868
869     /* Generic hardware linkage */
870     .irq =              uhci_irq,
871     .flags =            HCD_USB11,
872
873     /* Basic lifecycle operations */
874     .reset =            uhci_init,
875     .start =            uhci_start,
876 #ifdef CONFIG_PM
877     .suspend =          uhci_suspend,
878     .resume =           uhci_resume,
879     .bus_suspend =      uhci_rh_suspend,
880     .bus_resume =       uhci_rh_resume,
881 #endif
882     .stop =             uhci_stop,
883
884     .urb_enqueue =      uhci_urb_enqueue,
885     .urb_dequeue =      uhci_urb_dequeue,
886
887     .endpoint_disable = uhci_hcd_endpoint_disable,
888     .get_frame_number = uhci_hcd_get_frame_number,
889
890     .hub_status_data =  uhci_hub_status_data,
891     .hub_control =      uhci_hub_control,
892 };

```

其实就是一堆指针，也没什么了不起。不过我得提醒你了，在咱们整个故事中也只有一个 `struct hc_driver` 变量：`uhci_driver`。以后凡是提到 `hc_driver`，指的就是 `uhci_driver`。`probe` 函数是 PCI 那边的接口，而 `hc_driver` 是 USB 这边的接口，这两概念怎么扯到一块去的呢？呵呵，`usb_hcd_pci_probe` 函数 67 行，看见没有，`driver` 在这里被赋值了，而等号右边那个 `id->driver_data` 又是什么？继续回去看，在 `uhci_pci_ids` 这张表里写得很清楚，`driver_data` 就是被赋值为 `&uhci_driver`，所以说一切都是因才有果的，不会无缘无故地出现一个变量。

继续看 `usb_create_hcd()`，1496 行，一个变态的数据结构还不够，还得来一个更变态的。`struct usb_hcd`，有一个 HCD 就得有这么一个结构体，也来自 `drivers/usb/core/hcd.h`：

```

58 struct usb_hcd {

```

```

59
60     /*
61      * housekeeping
62      */
63     struct usb_bus      self;          /* hcd is-a bus */
64     struct kref          kref;         /* reference counter */
65
66     const char           *product_desc; /* product/vendor string */
67     char                 irq_descr[24]; /* driver + bus # */
68
69     struct timer_list    rh_timer;     /* drives root-hub polling */
70     struct urb           *status_urb;   /* the current status urb */
71 #ifdef CONFIG_PM
72     struct work_struct    wakeup_work;  /* for remote wakeup */
73 #endif
74
75     /*
76      * hardware info/state
77      */
78     const struct hc_driver *driver;     /* hw-specific hooks */
79
80     /* Flags that need to be manipulated atomically */
81     unsigned long         flags;
82 #define HCD_FLAG_HW_ACCESSIBLE 0x00000001
83 #define HCD_FLAG_SAW_IRQ      0x00000002
84
85     unsigned             rh_registered:1; /* is root hub registered? */
86
87     /* The next flag is a stopgap, to be removed when all the HCDs
88      * support the new root-hub polling mechanism. */
89     unsigned             uses_new_polling:1;
90     unsigned             poll_rh:1;      /* poll for rh status? */
91     unsigned             poll_pending:1; /* status has changed? */
92     unsigned             wireless:1;     /* Wireless USB HCD */
93
94     int                 irq;            /* irq allocated */
95     void __iomem         *regs;         /* device memory/io */
96     u64                 rsrc_start;     /* memory/io resource start */
97     u64                 rsrc_len;      /* memory/io resource length */
98     unsigned             power_budget;  /* in mA, 0 = no limit */
99
100 #define HCD_BUFFER_POOLS      4
101     struct dma_pool        *pool [HCD_BUFFER_POOLS];
102
103     int                 state;
104 #define __ACTIVE              0x01
105 #define __SUSPEND            0x04
106 #define __TRANSIENT          0x80
107
108 #define HC_STATE_HALT        0
109 #define HC_STATE_RUNNING    (__ACTIVE)
110 #define HC_STATE_QUIESCING  (__SUSPEND|__TRANSIENT|__ACTIVE)
111 #define HC_STATE_RESUMING   (__SUSPEND|__TRANSIENT)
112 #define HC_STATE_SUSPENDED  (__SUSPEND)
113
114 #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
115 #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
116
117     /* more shared queuing code would be good; it should support

```

```

118     * smarter scheduling, handle transaction translators, etc;
119     * input size of periodic table to an interrupt scheduler.
120     * (ohci 32, uhci 1024, ehci 256/512/1024).
121     */
122
123     /* The HC driver's private data is stored at the end of
124     * this structure.
125     */
126     unsigned long hcd_priv[0]
127         __attribute__((aligned (sizeof(unsigned long))));
128 };

```

所以 `usb_create_hcd` 函数就是用来为 `struct usb_hcd` 申请内存空间的，并且初始化。我们来看它具体如何初始化的。

1498 行，申请内存，并且初值为 0。

接下来得注意了，`usb_create_hcd` 中的 `dev` 可是 `struct device` 结构体指针，而刚才的 `usb_hcd_pci_probe` 中的 `dev` 是 `struct pci_dev` 结构体指针，`struct pci_dev` 表示的就是一个 PCI 设备，它有一个成员 `struct device dev`，所以实际上在调用 `usb_create_hcd` 时第二个参数是 `&dev->dev`。而这里 1503 行的 `dev_set_drvdata` 就是一简单的内联函数，来自 `include/linux/device.h`：

```

491 static inline void
492 dev_set_drvdata (struct device *dev, void *data)
493 {
494     dev->driver_data = data;
495 }

```

这个结构体中有一个成员 `void *driver_data`，其效果就是令 `dev->driver_data` 等于申请好的 `hcd`。

而 1504 行，初始化一个引用计数，可以看到 `struct usb_hcd` 有一个成员 `struct kref kref`，即引用计数的变量。

1506 行，`struct usb_hcd` 中有一个成员 `struct usb_bus self`，一个主机控制器就意味着一条总线，所以这里又出来另一个结构体：`struct usb_bus`。

```

276 struct usb_bus {
277     struct device *controller;    /* host/master side hardware */
278     int busnum;                  /* Bus number (in order of reg) */
279     char *bus_name;              /* stable id (PCI slot_name etc) */
280     u8 uses_dma;                  /* Does the host controller use DMA? */
281     u8 otg_port;                  /* 0, or number of OTG/HNP port */
282     unsigned is_b_host:1;        /* true during some HNP roleswitches */
283     unsigned b_hnp_enable:1;     /* OTG: did A-Host enable HNP? */
284
285     int devnum_next;              /* Next open device number in
286                                     * round-robin allocation */
287
288     struct usb_devmap devmap;    /* device address allocation map */

```



```

289     struct usb_device *root_hub;      /* Root hub */
290     struct list_head bus_list;        /* list of busses */
291
292     int bandwidth_allocated;          /* on this bus: how much of the time
293                                     * reserved for periodic (intr/iso)
294                                     * requests is used, on average?
295                                     * Units: microseconds/frame.
296                                     * Limits: Full/low speed reserve 90%,
297                                     * while high speed reserves 80%.
298                                     */
299     int bandwidth_int_reqs;           /* number of Interrupt requests */
300     int bandwidth_isoc_reqs;         /* number of Isoc. requests */
301
302 #ifdef CONFIG_USB_DEVICEFS
303     struct dentry *usbfs_dentry; /* usbfs dentry entry for the bus */
304 #endif
305     struct class_device *class_dev; /* class device for this bus */
306
307 #if defined(CONFIG_USB_MON)
308     struct mon_bus *mon_bus;         /* non-null when associated */
309     int monitored;                   /* non-zero when monitored */
310 #endif
311 };

```

## 4. I/O 内存和 I/O 端口

`usb_bus_init` 来自 `drivers/usb/core/hcd.c`，很显然，它就是初始化 `struct usb_bus` 结构体指针。而这个结构体变量 `hcd->self` 的内存已经在刚才为 HCD 申请内存时一并申请了。

```

695 static void usb_bus_init (struct usb_bus *bus)
696 {
697     me_mset (&bus->devmap, 0, sizeof(struct usb_devmap));
698
699     bus->devnum_next = 1;
700
701     bus->root_hub = NULL;
702     bus->busnum = -1;
703     bus->bandwidth_allocated = 0;
704     bus->bandwidth_int_reqs = 0;
705     bus->bandwidth_isoc_reqs = 0;
706
707     INIT_LIST_HEAD (&bus->bus_list);
708 }

```

当初在 Hub 驱动中就讲过，`devnum_next` 在总线初始化时会被设为 1，说的就是这里。

回到 `usb_create_hcd` 中来，又是几行赋值。

倒是 1511 行引起了我的注意，又是“可恶”的时间机制：`init_timer`，这个函数我们也见过多次了，斑驳的陌生终于在时间的抚摸下变成了今日的熟悉。这里设置的函数是

rh\_timer\_func，而传递给这个函数的参数是 hcd。

1515 行，INIT\_WORK 也在 Hub 驱动里见过了，这里 hcd\_resume\_work 什么时候会被调用也到时候再看。

剩下两行赋值。1518 行，struct usb\_hcd 有一个 struct hc\_driver 的结构体指针成员，所以就把它和咱们这个 uhci\_driver 给联系起来了。而在 uhci\_driver 中可看到，其中有一个 product\_desc 被赋值为“UHCI Host Controller”，所以这里也赋给 hcd->product\_desc，因为 struct hc\_driver 和 struct usb\_hcd 这两个结构体中都有一个成员 const char \*product\_desc。

至此，usb\_create\_hcd 结束了，返回了申请好赋好值的 hcd。我们继续回到 probe 函数中来。

89 到 124 这个 if-else 的确让我开心，因为 if 里说的是 EHCI 和 OHCI 的情况，else 里针对的才是 UHCI，所以这就意味着这个 if-else 只要从 105 行开始看，即直接看到 UHCI 那部分的代码。

不过也得知道这里为何要判断 HCD\_MEMORY，这个宏是表明该 HC 的寄存器是使用 Memory 的，而没有设置 flag 的 HC 的寄存器是使用 I/O 的。这些寄存器俗称 I/O 端口(I/O ports)，这个 I/O 端口可以被映射在 Memory Space，也可以被映射在 I/O Space。UHCI 是属于后者，而 EHCI/OHCI 属于前者。

这里看上去必须多说几句，否则很难说清楚。以我们家 Intel 为代表的 i386 系列处理器中，内存和外部 I/O 是独立编址、独立寻址的，于是有一个地址空间叫做内存空间，另有一个地址空间叫做 I/O 空间。也就是说，从处理器的角度来说，i386 提供了一些单独的指令用来访问 I/O 空间。换言之，访问 I/O 空间和访问普通的内存得使用不同的指令。而在一些嵌入式处理器中，比如 PowerPC 只使用一个空间，即内存空间。那像这种情况，外设的 I/O 端口的物理地址就被映射到内存地址空间中，这就是内存映射（Memory-mapped）。而外设的 I/O 端口的物理地址就被映射到 I/O 地址空间中，这就是 I/O-mapped，即 I/O 映射。

那么 EHCI/OHCI 除了有寄存器以外，还有内存。而它们把这些统统映射到内存空间中去，而 UHCI 只用寄存器来通信，所以它只需要映射寄存器，即 I/O 端口。而 spec 规定，它是映射到 I/O 空间的。Linux 中 I/O 内存和 I/O 端口都被视为一种资源，分别被记录在 /proc/iomem 和 /proc/ioports 中。

所以可以在这里看到 uhci-hcd:

```
localhost:~ # cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
□□
bca0-bcbf : 0000:00:1d.2
    bca0-bcbf : uhci_hcd
bcc0-bcdf : 0000:00:1d.1
```

```

bcc0-bcdf : uhci_hcd
bce0-bcff : 0000:00:1d.0
bce0-bcff : uhci_hcd
c000-cfff : PCI Bus #10
cc00-ccff : 0000:10:0d.0
d000-dfff : PCI Bus #0e
dcc0-dcdf : 0000:0e:00.1
dcc0-dcdf : e1000
dce0-dcff : 0000:0e:00.0
dce0-dcff : e1000
e000-ffff : PCI Bus #0c
e800-e8ff : 0000:0c:00.1
e800-e8ff : qla2xxx
ec00-ecff : 0000:0c:00.0
ec00-ecff : qla2xxx
fc00-fc0f : 0000:00:1f.1
fc00-fc07 : ide0

```

而在这里看到 ehci-hcd:

```

localhost:~ # cat /proc/iomem
00000000-0009ffff : System RAM
00000000-00000000 : Crash kernel
□□
d8000000-d80fffff : PCI Bus #01
d8000000-d80fffff : PCI Bus #02
d80f0000-d80fffff : 0000:02:0e.0
d80f0000-d80fffff : megasas: LSI Logic
d8100000-d81fffff : PCI Bus #0c
d8100000-d813ffff : 0000:0c:00.1
e0000000-efffffff : reserved
f2000000-f7ffffff : PCI Bus #06
f4000000-f7ffffff : PCI Bus #07
f4000000-f7ffffff : PCI Bus #08
f4000000-f7ffffff : PCI Bus #09
f4000000-f5ffffff : 0000:09:00.0
f4000000-f5ffffff : bnx2
f8000000-fbffffff : PCI Bus #04
f8000000-fbffffff : PCI Bus #05
f8000000-f9ffffff : 0000:05:00.0
f8000000-f9ffffff : bnx2
□□
fca00400-fca007ff : 0000:00:1d.7
fca00400-fca007ff : ehci_hcd
fe000000-ffffffff : reserved
100000000-22ffffff : System RAM

```

使用 I/O 内存要先申请，再映射。使用 I/O 端口也要先申请，也可称为请求，即让内核知道你要访问这个端口，这样内核知道了以后它就不会再让别人也访问这个端口了。申请 I/O 端口的函数是 `request_region`，这个函数来自 `include/linux/ioport.h`:

```

116 /* Convenience shorthand with allocation */
117 #define request_region(start,n,name) \
    __request_region(&ioport_resource, (start), (n), (name))
118 #define request_mem_region(start,n,name) \
    __request_region(&iomem_resource, (start), (n), (name))

```

```

119 #define rename_region(region, newname) \
    do { (region)->name = (newname); } while (0)
120
121 extern struct resource * __request_region(struct resource *,
122     resource_size_t start,
123     resource_size_t n, const char *name);

```

这里看到的 `request_mem_region` 用来申请 I/O 内存。申请了之后，还需要使用 `ioremap` 或者 `ioremap_nocache` 函数来映射。

`request_region` 的三个参数 `start`, `n`, `name` 意味着你想使用从 `start` 开始的 `size` 为 `n` 的 I/O port 资源，`name` 自然就是你的名字了。这三个概念在 `cat /proc/ioprots` 的里面显示得很清楚，`name` 就是 `uhci-hcd`。

那么对于 `uhci-hcd`，我们究竟需要请求哪些地址，需要多少空间呢？嗯，又要提到那张“上坟图”了。PCI 设备本身有一堆的地址空间、内存空间和 I/O 空间。那么用什么把这些空间映射到总线上来呢？寄存器。每个设备都有 6 个地址空间，这叫做 6 个基址寄存器，有的设备还有一个 ROM，所以又有一个 Expansion ROM Base Address，它对应第 7 个区间，或者说区间 6，而在“上坟图”上对应的就叫做扩展 ROM 基址寄存器。每个寄存器都是 4 个字节。而在 `include/linux/pci.h` 中定义了：

```

235 #define PCI_ROM_RESOURCE 6

```

所以看到循环条件就是从 0 到 `PCI_ROM_RESOURCE` 之前，即循环 6 次，因为有 6 个区间，区间也叫 `region`。那么这些寄存器究竟取的什么值呢？这就是在 PCI 总线初始化时做的事情了，它会把每个基址寄存器赋上值，而实际上就是映射于总线上的地址，总线驱动的作用就是让各个设备需要的地址资源都得到满足，并且设备与设备之间的地址不发生冲突。PCI 总线驱动做了这些之后，我们 PCI 设备驱动就简单了，在需要使用时直接请求即可，正如这里的 `request_region`。那么传递给 `request_region` 的具体参数是什么呢？

两个函数，`pci_resource_start` 和 `pci_resource_len`，去获得一个区间的起始地址和长度，所以就很好理解这段代码了。至于 109 行这个 `if` 判断，`pci_resource_flags` 是用来判断一个资源是哪种类型，`include/linux/ioport.h` 中一共定义了 4 种资源：

```

36 #define IORESOURCE_IO          0x00000100    /* Resource type */
37 #define IORESOURCE_MEM        0x00000200
38 #define IORESOURCE_IRQ        0x00000400
39 #define IORESOURCE_DMA        0x00000800

```

它们是 I/O、内存、中断和 DMA。对应应在 `/proc` 下的 `ioprots`, `iomem`, `interrupt` 和 `dma` 4 个文件。所以这里就判断如果不是 I/O 端口资源，那么就不予理睬。因为 UHCI 主机控制器只需要理睬 I/O 端口。

`request_region` 函数执行成功将返回非 `NULL`，否则返回 `NULL`。所以一旦成功就跳出循环。反之，如果循环都结束了还未能请求到，那就说明出错了。那么为何一旦成功就跳出循环？老

实说，这个问题足足困扰了我 13 秒钟，别小看这 13 秒钟，有这么长时间刘翔都已经完成一次 110 米跨栏了。让 spec 来告诉你。

看到没有，20~23h，4 个字节，这里正好对应 UHCI 的 I/O 空间基址寄存器。换言之，UHCI 就定义了一个基址寄存器，所以只要使用一个基址寄存器就可以映射所需要的地址了。所以，成功一次就可以结束循环了。

又一次回到 `usb_hcd_pci_probe` 中，126 行，`pci_set_master` 函数。还是看那张“上坟图”，注意到第三个寄存器，即命令寄存器。让我们用 PCI spec 来告诉你这个命令寄存器的格局。

看到 Bus Master 了么？没错，就是那个 Bit 2。用毛德操先生的话说就是“PCI 设备要进行 DMA 操作就得具有竞争成为总线主的能力”。而这个 Bus Master 位就是用来打开或关闭 PCI 设备竞争成为总线主的能力的。在完成 PCI 总线的初始化时，所有 PCI 设备的 DMA 功能都是关闭的，所以这里要调用 `pci_set_master` 启用 USB 主机控制器竞争成为总线主的能力。就是说 PCI 设备有没有这种能力是可以设置的。

USB spec 2.0 中 10.2.9 节在讲到 USB Host Interface 时，说了 USB HC 是应该具备这种能力的：“The Host Controller provides a high-speed bus-mastering interface to and from main system memory. The physical transfer between memory and the USB wire is performed automatically by the Host Controller.”

同时在 UHCI spec 里面我们也能找到这么一句话：“For the implementation example in this document, the Host Controller is a PCI device. PCI Bus master capability in the Host Controller permits high performance data transfers to system memory.”

不过，究竟什么是 PCI 的 Bus Master？连接到 PCI 总线上的设备有两种：主控设备和目标设备，即 master 设备和 target-only 设备。这两者最直接的区别就是 target-only 最少需要 47 根 pin，而 master 最少需要 49 根 pin，它们所必须支持的总线信号就是不一样的。

PCI 设备如果以 target-only 的方式工作，那么它就完全是在主机的 CPU 的控制之下工作，比如设备接收到某一个外部事件，中断主机，主机 CPU 读写设备，这样设备就可以工作了。这样的设备也被少数人称为 slave 设备，或者说从设备。这就是典型的奴才型的设备，主说什么就是什么，完全没有自己的见解。而 master 设备就比这个要复杂了，master 设备能够不在主机 CPU 的干预下访问主机的地址空间，包括主存和其他 PCI 设备。很显然，DMA 就属于这种情况，即不需要主机 CPU 干涉的情况下 USB 主机控制器通过 DMA 直接读写内存。所以需要启用这种能力。

不过 PCI 总线在同一时刻只能供一对设备完成传输。至于有了竞争力能不能竞争得到 Master，那就得看人品了。上天给了你一幅天使的面孔和魔鬼的身材，但你能不能成为明星就得看造化了，当然只要你遵守圈中的潜规则，你离成功就不远了。

## 5. 传说中的 DMA

下一个函数，usb\_add\_hcd，定义在 drivers/usb/core/hcd.c 中：

```

1558 int usb_add_hcd(struct usb_hcd *hcd,
1559                 unsigned int irqnum, unsigned long irqflags)
1560 {
1561     int retval;
1562     struct usb_device *rhdev;
1563
1564     dev_info(hcd->self.controller, "%s\n", hcd->product_desc);
1565     set_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags);
1566
1567     /* HC is in reset state, but accessible. Now do the one-time init,
1568      * bottom up so that hcids can customize the root hubs before khubd
1569      * starts talking to them. (Note, bus id is assigned early too.)
1570      */
1571     if ((retval = hcd_buffer_create(hcd)) != 0) {
1572         dev_dbg(hcd->self.controller, "pool alloc failed\n");
1573         return retval;
1574     }
1575
1576     if ((retval = usb_register_bus(&hcd->self)) < 0)
1577         goto err_register_bus;
1578
1579     if ((rhdev = usb_alloc_dev(NULL, &hcd->self, 0)) == NULL) {
1580         dev_err(hcd->self.controller, "unable to allocate root hub\n");
1581         retval = -ENOMEM;
1582         goto err_allocate_root_hub;
1583     }
1584
1585     rhdev->speed = (hcd->driver->flags & HCD_USB2) ? USB_SPEED_HIGH :
1586                 USB_SPEED_FULL;
1587     hcd->self.root_hub = rhdev;
1588
1589     /* wakeup flag init defaults to "everything works" for root hubs,
1590      * but drivers can override it in reset() if needed, along with
1591      * recording the overall controller's system wakeup capability.
1592      */
1593     device_init_wakeup(&rhdev->dev, 1);
1594
1595     /* "reset" is misnamed; its role is now one-time init. the controller
1596      * should already have been reset (and boot firmware kicked off etc).
1597      */
1598     if (hcd->driver->reset && (retval = hcd->driver->reset(hcd)) < 0) {
1599         dev_err(hcd->self.controller, "can't setup\n");
1600         goto err_hcd_driver_setup;
1601     }
1602
1603     /* NOTE: root hub and controller capabilities may not be the same */
1604     if (device_can_wakeup(hcd->self.controller)
1605         && device_can_wakeup(&hcd->self.root_hub->dev))
1606         dev_dbg(hcd->self.controller, "supports USB remote wakeup\n");
1607
1608     /* enable irqs just before we start the controller */
1609     if (hcd->driver->irq) {
1610         snprintf(hcd->irq_descr, sizeof(hcd->irq_descr), "%s:usb%d",

```

```

1611             hcd->driver->description, hcd->self.busnum);
1612     if ((retval = request_irq(irqnum, &usb_hcd_irq, irqflags,
1613         hcd->irq_descr, hcd)) != 0) {
1614         dev_err(hcd->self.controller,
1615             "request interrupt %d failed\n", irqnum);
1616         goto err_request_irq;
1617     }
1618     hcd->irq = irqnum;
1619     dev_info(hcd->self.controller, "irq %d, %s 0x%08llx\n", irqnum,
1620         (hcd->driver->flags & HCD_MEMORY) ?
1621         "io mem" : "io base",
1622         (unsigned long long)hcd->rsrc_start);
1623 } else {
1624     hcd->irq = -1;
1625     if (hcd->rsrc_start)
1626         dev_info(hcd->self.controller, "%s 0x%08llx\n",
1627             (hcd->driver->flags & HCD_MEMORY) ?
1628             "io mem" : "io base",
1629             (unsigned long long)hcd->rsrc_start);
1630 }
1631
1632 if ((retval = hcd->driver->start(hcd)) < 0) {
1633     dev_err(hcd->self.controller, "startup error %d\n", retval);
1634     goto err_hcd_driver_start;
1635 }
1636
1637 /* starting here, usbcore will pay attention to this root hub */
1638 rhdev->bus_mA = min(500u, hcd->power_budget);
1639 if ((retval = register_root_hub(hcd)) != 0)
1640     goto err_register_root_hub;
1641
1642 if (hcd->uses_new_polling && hcd->poll_rh)
1643     usb_hcd_poll_rh_status(hcd);
1644 return retval;
1645
1646 err_register_root_hub:
1647     hcd->driver->stop(hcd);
1648 err_hcd_driver_start:
1649     if (hcd->irq >= 0)
1650         free_irq(irqnum, hcd);
1651 err_request_irq:
1652 err_hcd_driver_setup:
1653     hcd->self.root_hub = NULL;
1654     usb_put_dev(rhdev);
1655 err_allocate_root_hub:
1656     usb_deregister_bus(&hcd->self);
1657 err_register_bus:
1658     hcd_buffer_destroy(hcd);
1659     return retval;
1660 }

```

1566 行，设置一个 flag，至于作用，等遇到了再说。

1572 行，hcd\_buffer\_create，初始化一个 buffer 池。现在是时候说一说 DMA 了。我们知道一个 USB 主机控制器控制着一条 USB 总线，而 USB 主机控制器的一项重要工作是在内存和 USB 总线之间传输数据。这个过程可以使用 DMA 也可以不使用 DMA，不使用 DMA 的方式即

所谓的 PIO 方式。DMA 代表着 Direct Memory Access，即直接内存访问。即不需要 CPU 干预，比如有一个 UHCI 控制器，我告诉它，内存中某个地方放了一堆数据，你去取吧，然后它就自己去取，取完了它就跟我说一声，告诉我它取完了。

那么在整个 USB 子系统中是如何处理这些事情的呢？好，苦等了这么久，我终于有机会来向你解释这个问题了，现在我终于可以说电视剧中男主角对女主角常说的那句话，“你听我解释，你听我解释呀！”在 Hub 驱动中，调用过一个函数 `usb_buffer_alloc`，当时很多人问我这个函数究竟是如何处理 DMA 或不 DMA 的？

关于 DMA，合理的做法是先创建一个内存池，然后每次都从池子里要内存。具体说就是先由 HCD 这边建池子，然后设备驱动就直接索取。来看一下来自 `drivers/usb/core/buffer.c` 中的 `hcd_buffer_create`：

```
52 int hcd_buffer_create(struct usb_hcd *hcd)
53 {
54     char          name[16];
55     int           i, size;
56
57     if (!hcd->self.controller->dma_mask)
58         return 0;
59
60     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
61         if (!(size = pool_max[i]))
62             continue;
63         snprintf(name, sizeof name, "buffer-%d", size);
64         hcd->pool[i] = dma_pool_create(name, hcd->self.controller,
65                                     size, size, 0);
66         if (!hcd->pool[i]) {
67             hcd_buffer_destroy(hcd);
68             return -ENOMEM;
69         }
70     }
71     return 0;
72 }
```

先看 64 行，调用 `dma_pool_create` 函数，这个函数是真正去创建内存池的函数，或者更准确地讲，创建一个 DMA 池，内核中定义了一个结构体：`struct dma_pool` 来专门代表一个 DMA 池。而这个函数的返回值就是生成的那个 DMA 池。如果创建失败就调用 `hcd_buffer_destroy`，还是来自同一个文件：

```
82 void hcd_buffer_destroy(struct usb_hcd *hcd)
83 {
84     int           i;
85
86     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
87         struct dma_pool *pool = hcd->pool[i];
88         if (pool) {
89             dma_pool_destroy(pool);
90             hcd->pool[i] = NULL;
91         }
92     }
```



看得出这里调用的是 `dma_pool_destroy`，其作用不言自明。

那么我们知道了创建池子和销毁池子的函数，如何从池子里索取或者把索取的释放回去呢？这对应的两个函数分别是，`dma_pool_alloc` 和 `dma_pool_free`，而这两个函数正是与我们说的 `usb_buffer_alloc` 以及 `usb_buffer_free` 相联系的。于是来看这两个函数的代码，来自 `drivers/usb/core/usb.c`：

```

589 void *usb_buffer_alloc(
590     struct usb_device *dev,
591     size_t size,
592     gfp_t mem_flags,
593     dma_addr_t *dma
594 )
595 {
596     if (!dev || !dev->bus)
597         return NULL;
598     return hcd_buffer_alloc(dev->bus, size, mem_flags, dma);
599 }
600
612 void usb_buffer_free(
613     struct usb_device *dev,
614     size_t size,
615     void *addr,
616     dma_addr_t dma
617 )
618 {
619     if (!dev || !dev->bus)
620         return;
621     if (!addr)
622         return;
623     hcd_buffer_free(dev->bus, size, addr, dma);
624 }
```

很显然，它们调用的就是 `hcd_buffer_alloc` 和 `hcd_buffer_free`，于是进一步跟踪，来自 `drivers/usb/core/buffer.c`：

```

100 void *hcd_buffer_alloc(
101     struct usb_bus *bus,
102     size_t size,
103     gfp_t mem_flags,
104     dma_addr_t *dma
105 )
106 {
107     struct usb_hcd *hcd = bus_to_hcd(bus);
108     int i;
109
110     /* some USB hosts just use PIO */
111     if (!bus->controller->dma_mask) {
112         *dma = ~(dma_addr_t) 0;
113         return kmalloc(size, mem_flags);
114     }
115
116     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
```

```

117         if (size <= pool_max [i])
118             return dma_pool_alloc(hcd->pool [i], mem_flags, dma);
119     }
120     return dma_alloc_coherent(hcd->self.controller, size, dma, 0);
121 }
122
123 void hcd_buffer_free(
124     struct usb_bus *bus,
125     size_t          size,
126     void            *addr,
127     dma_addr_t      dma
128 )
129 {
130     struct usb_hcd *hcd = bus_to_hcd(bus);
131     int i;
132
133     if (!addr)
134         return;
135
136     if (!bus->controller->dma_mask) {
137         kfree(addr);
138         return;
139     }
140
141     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
142         if (size <= pool_max [i]) {
143             dma_pool_free(hcd->pool [i], addr, dma);
144             return;
145         }
146     }
147     dma_free_coherent(hcd->self.controller, size, addr, dma);
148 }

```

看见了吧，最终调用的就是 `dma_pool_alloc` 和 `dma_pool_free`。那么主机控制器到底支不支持 DMA 操作呢？看见上面这个 `dma_mask` 了么？默认情况下，`dma_mask` 在总线枚举时被函数 `pci_scan_device` 中设置为了 `0xffffffff`。`struct device` 结构体有一个成员 `u64 *dma_mask`，如果一个 PCI 设备不能支持 DMA，那么应该在 `probe` 函数中调用 `pci_set_dma_mask` 把这个 `dma_mask` 设置为 `NULL`。不过一个没有精神分裂症的 PCI 设备通常是支持 DMA 的。这个掩码更多的作用是，比如你的设备只能支持 24 位的寻址，那你就得通过设置 `dma_mask` 来告诉 PCI 层，就需要把 `dma_mask` 设置为 `0x00ffffff`。因为标准的 PCI 设备都是 32 位寻址的，所以标准情况就是设置的 `0xffffffff`。不过开发人员们的建议是不要直接使用这些数字，而是使用它们定义在 `include/linux/dma-mapping.h` 中的这些宏：

```

16 #define DMA_64BIT_MASK 0xffffffffffffffffULL
17 #define DMA_48BIT_MASK 0x0000ffffffffffffULL
18 #define DMA_40BIT_MASK 0x000000ffffffffffffULL
19 #define DMA_39BIT_MASK 0x0000007ffffffffffffULL
20 #define DMA_32BIT_MASK 0x00000000ffffffffULL
21 #define DMA_31BIT_MASK 0x000000007ffffffffULL
22 #define DMA_30BIT_MASK 0x000000003ffffffffULL
23 #define DMA_29BIT_MASK 0x000000001ffffffffULL
24 #define DMA_28BIT_MASK 0x000000000ffffffffULL
25 #define DMA_24BIT_MASK 0x000000000ffffffffULL

```

不过目前在 `drivers/usb/` 目录下面没有哪个驱动会调用 `pci_set_dma_mask`，因为现代总线上的大部分设备都能够处理 32 位地址，换句话说，大家的设备都还算“正经”，但如果你们家生产出来一个“不伦不类”的设备，那么你就别忘了在 `probe` 阶段用 `pci_set_dma_mask` 设置一下，否则你就甭指望设备能够正确进行 DMA 传输。关于这个函数的使用，可以参考 `drivers/net` 下面的那些驱动，很多网卡驱动都调用了这个函数，虽然其中很多其实就是设置 32 位。

要不，总结一下？以上这几个 DMA 函数就不细讲了。但需要对某些地方单独拿出来讲。

第一，`hcd_buffer_alloc` 函数中，111 行，判断：如果 `dma_mask` 为 `NULL`，说明这个主机控制器不支持 DMA，那么使用原始的方法申请内存，即 `kmalloc`。然后申请好了就直接返回，而不会继续去执行下面的 `dma_pool_alloc` 函数。同样的判断在 `hcd_buffer_free` 中也是一样的，没有 DMA 的就直接调用 `kfree` 释放内存，而不需要调用 `dma_pool_free` 了。

第二，你应该注意到这里还有另外两个函数我们根本没提起：`dma_alloc_coherent` 和 `dma_free_coherent`。这两个函数也是用来申请 DMA 内存的，但是它们适合申请比较大的内存，比如 N 个 `page` 的那种。而 DMA 池的作用本来就是提供给“小打小闹”式的内存申请的。当前的 USB 子系统里在 `drivers/usb/core/buffer.c` 中定义了一个数组：

```
26 static const size_t    pool_max [HCD_BUFFER_POOLS] = {
27     /* platfor ms without dma-friendly caches might need to
28      * prevent cacheline sharing...
29      */
30     32,
31     128,
32     512,
33     PAGE_SIZE / 2
34     /* bigger --> allocate pages */
35 };
```

`HCD_BUFFER_POOLS` 这个宏的值为 4。结合 `hcd_buffer_alloc` 函数中面那个循环来看，可以知道，你要是申请 32 个字节以内、128 个字节以内、512 个字节以内，或者最多二分之一一个 `PAGE_SIZE` 以内的，就直接使用这个内存池了。否则的话，就得用那个 `dma_alloc_coherent` 了。释放时也一样。

第三，`struct usb_hcd` 结构体有这么一个成员，

```
struct dma_pool    *pool [HCD_BUFFER_POOLS]
```

这个数组的值是在 `hcd_buffer_create` 中赋上的，即当时以这个 `pool_max` 为模型创建了 4 个池子，所以 `hcd_buffer_alloc` 里就可以这样用。

第四，至于像 `dma_pool_create/dma_pool_alloc/dma_alloc_coherent` 这些函数具体怎么实现的，我想任何一个写设备驱动程序的都不用关心吧，倒是有人会比较感兴趣，因为他们是研究内存管理的。

OK，讲完了 hcd\_buffer\_create，让我们还是回到 usb\_add\_hcd 中来，继续往下走。

## 6. 来来，我是一条总线，线线线线线线

请注意，下一个函数，1577 行，usb\_register\_bus()。我们说过，一个 USB 主机控制器就意味着一条 USB 总线，因为主机控制器控制的正是一条总线。古人说，猫走不走直线，完全取决于耗子，而数据走不走总线，完全取决于主机控制器。

所以这里作为主机控制器的驱动，我们必须从软件的角度来说，注册一条总线。来自 drivers/usb/core/hcd.c:

```

720 static int usb_register_bus(struct usb_bus *bus)
721 {
722     int busnum;
723
724     mutex_lock(&usb_bus_list_lock);
725     busnum = find_next_zero_bit (busmap.busmap, USB_MAXBUS, 1);
726     if (busnum < USB_MAXBUS) {
727         set_bit (busnum, busmap.busmap);
728         bus->busnum = busnum;
729     } else {
730         printk (KERN_ERR "%s: too many buses\n", usbcore_name);
731         mutex_unlock(&usb_bus_list_lock);
732         return -E2BIG;
733     }
734
735     bus->class_dev=class_device_create(usb_host_class,NULL,MKDEV(0,0),
736                                     bus->controller, "usb_host%d", busnum);
737     if (IS_ERR(bus->class_dev)) {
738         clear_bit(busnum, busmap.busmap);
739         mutex_unlock(&usb_bus_list_lock);
740         return PTR_ERR(bus->class_dev);
741     }
742
743     class_set_devdata(bus->class_dev, bus);
744
745     /* Add it to the local list of buses */
746     list_add (&bus->bus_list, &usb_bus_list);
747     mutex_unlock(&usb_bus_list_lock);
748
749     usb_notify_add_bus(bus);
750
751     dev_info (bus->controller,
752             "new USB bus registered, assigned bus number %d\n", bus->busnum);
753     return 0;
754 }
```

Linux 中名字里带一个 register 的函数那是数不胜数，随着你对 Linux 内核渐渐熟悉，慢慢就会觉得其实叫 register 的函数都很简单。甚至你会发现，Linux 内核中的模块没有不用 register

函数的。

这个函数首先让我们想起了在 Hub 驱动中讲的那个 `choose_address`。当时有一个 `devicemap`，而现在有一个 `busmap`。很显然，原理是一样的。在 `drivers/usb/core/hcd.c` 中有定义：

```
89 #define USB_MAXBUS          64
90 struct usb_busmap {
91     unsigned long busmap [USB_MAXBUS / (8*sizeof (unsigned long))];
92 };
93 static struct usb_busmap busmap;
```

和当时我们在 Hub 驱动中对 `devicemap` 的分析一样，当时的结论是该 `map` 一共有 128 位，同理可知这里 `busmap` 则一共有 64 位。也就是说一共可以有 64 条 USB 总线。我想，对我们这些凡夫俗子来说，这么多条足够了把。

735 行，`class_device_create` 函数是 Linux 设备模型中一个很基础的函数。Intel 的企业文化中有六大价值观（去 Intel 面试时我特逗的一点就是把那六大价值观给背了下来，然后面试时跟面试官一条一条说，把人家逗乐了），这六大价值观中有一个叫做 **Result Orientation**，用中文说就是以结果为导向。那现在我想是该使用这一价值观的时候了，Linux 2.6 设备模型中提供了大把大把的基础函数，它们都在 `drivers/base` 目录下面，这里的函数你如果有兴趣当然可以看一看，不过我不推荐你这么，除非你的目的就是要彻底研究这个设备模型是如何实现的。对这些基础函数，我觉得比较好的认识方法就是以结果为导向，查看它们执行之后具体有什么效果，用直观的效果来体现它们的作用。

那么这里 `class_device_create` 的效果是什么？我们知道设备模型和 `sysfs` 结合相当紧密，最能反映设备模型的效果的就是 `sysfs`。所以凭一种男人的直觉，我们应该到 `sysfs` 下面去看效果。不过我现在更愿意给你提供更多的背景知识。

首先，什么是 `class`？C++ 的高手们一定不会不知道 `class` 吧，虽说我从未写过 C++ 程序，可是好歹也看过两遍《Thinking in C++》的第一卷，所以 `class` 还是知道的。`class` 就是类，设备模型中引入类的意义在于让一些模糊的东西变得更加清晰、直观，比如同样是 SCSI 设备，可能磁盘和磁带，都属于 `scsi_device`。于是可以在 `/sys/class` 下面建立一个文件夹，从这里来体现同一个类别的各种设备。比如，某台机器里 `/sys/class` 下面可以看到这些类，此时此刻还没有加载 `usbcore`：

```
localhost:~ # ls /sys/class/
backlight graphics mem net spi_master vc
dma input misc pci_bus tty vtconsole
```

而 `class_device_create` 的第一个参数是 `usb_host_class`，它是什么呢？

让我们把镜头切给 `usbcore` 的初始化函数，`usb_init()`。在 Hub 驱动中已经说过，`usb_init` 是 Linux 中整个 USB 子系统的起点，一切的一切都在这里开始。而这个函数中有这么一段：

```

877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;

```

来看它具体做了什么事情，usb\_host\_init()来自 drivers/usb/core/hcd.c:

```

671 static struct class *usb_host_class;
672
673 int usb_host_init(void)
674 {
675     int retval = 0;
676
677     usb_host_class = class_create(THIS_MODULE, "usb_host");
678     if (IS_ERR(usb_host_class))
679         retval = PTR_ERR(usb_host_class);
680     return retval;
681 }

```

让我们在上面提到的那台机器中加载 usbcore，查看在/sys/class/下面会发生点什么：

```

localhost:~ # modprobe usbcore
localhost:~ # ls /sys/class/
backlight  dma  graphics  input  mem  misc  net  pci_bus  spi_master  tty
usb_device  usb_host  vc  vtconsole

```

看出区别了么？多了两个目录，usb\_host 和 usb\_device，换言之，多了两个类。所以不难知道，这里 class\_create 函数的效果就是在/sys/class/下面创建一个 usb\_host 的目录。而 usb\_device 是在另一个函数 usb\_devio\_init()中创建的。创建方法是一样的，均是调用 class\_create 函数。

继续看 usb\_host，调用 class\_create 将返回值赋给了 usb\_host\_class，而这正是我们传递给 class\_device\_create 的第一个参数，所以不看代码也应该知道我们的目标是在/sys/class/usb\_host/下面建立一个文件或者一个目录，那么结合代码来看就不难发现我们要建立的是一个叫做 usb\_hostn 的文件或者是目录，具体是什么，让我们用结果来说话。没有加载 uhci-hcd 时，可以看出这个目录是空的。

```

localhost:~ # ls /sys/class/usb_host/

```

把这个模块加载，再来查看效果：

```

localhost:~ # modprobe uhci-hcd
localhost:~ # ls -l /sys/class/usb_host/
total 0
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host1
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host2
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host3
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host4

```

因为这台机器有 4 个 UHCI 主机控制器，所以可以看出，分别为每个主机控制器建立了一个目录。而 usb\_host 后面的这个 1, 2, 3, 4 就是刚才说的 busnum，即总线编号，因为一个主机控制器控制着一条总线。

同时我们把 class\_device\_create 的返回值赋给了 bus->class\_dev。struct usb\_bus 中有一个成

员 `struct class_device *class_dev`，这个成员被称作 `class device`。这个结构体对写驱动的人来说意义不大，但是从设备模型的角度来说是必要的，实际上对写驱动的人来说，你完全可以不理睬设备模型中 `class` 这个部分，可以尽可能少地支持设备模型，因为这对访问设备没有太多影响。甚至可以让设备根本就不在 `/sysfs` 下面体现出来，用代码去支持设备模型，将来你使用设备时就能享受到设备模型为你提供的方便。相反，如果不支持设备模型，那么使用设备时会发现有很多不便。所以 743 行做了这么一个举动，调用 `class_set_devdata()`，这就算是写代码的人对设备模型的支持，因为 `struct class_device` 中也有一个成员 `void *class_data`，被称为 `class-specific data`，而在 `include/linux/device.h` 中定义了 `class_set_devdata()` 和一个与之对应的函数 `class_get_devdata()`。

```
279 static inline void *
280 class_get_devdata (struct class_device *dev)
281 {
282     return dev->class_data;
283 }
284
285 static inline void
286 class_set_devdata (struct class_device *dev, void *data)
287 {
288     dev->class_data = data;
289 }
```

结合这里具体对这个函数调用的代码可知，最终这个主机控制器对应的 `class_device` 的 `class_data` 被赋值为 `bus`。这样有朝一日要通过 `class_device` 找到对应的 `bus` 时只要调用 `class_get_devdata` 即可。设备模型的精髓就在于把一个设备相关联的种种元素都给联系起来，设备模型提供了大量建立这种纽带的函数。我们要做的就是调用这些函数。

好，继续，746 行，很显然又是队列操作。`usb_bus_list` 是一个全局队列，在 `drivers/usb/core/hcd.c` 中定义：

```
85 LIST_HEAD (usb_bus_list);
86 EXPORT_SYMBOL_GPL (usb_bus_list);
```

每次注册一条总线就是往这个队列里添加一个元素。`struct usb_bus` 中有一个成员 `struct list_head bus_list`。所以这里直接调用 `list_add` 即可。

用一句话解释 749 行，`usb_notify_add_bus`，按 Intel 以结果为导向的理论来说，这个函数在此情此景执行的结果是 `/proc/bus/usb` 下面会多一些文件，比如：

```
localhost:~ # ls /proc/bus/usb/
001 002 003 004 devices
```

好了，`usb_register_bus` 算是看完了，再一次回到 `usb_add_hcd` 中来，1580 行，`usb_alloc_dev` 被调用，这个函数我们可不陌生。不过我们下节再看吧。这节剩下的时间我们为 `usb_notify_add_bus` 再多说两句话，这个函数牵涉到 Linux 中的 `notify` 机制，这是 Linux 内核中一种常用的事件回调处理机制。传说中的那个“神奇”的内核调试工具 `KDB` 中就是利用这种

机制进入 KDB 的。

这种机制在网络设备驱动中的应用，那就像“成都小吃”在北京满大街都是一样。而在 USB 子系统中，以前并没有使用这种机制，只是 Linux 中 USB 掌门人 Greg 在 2005 年底提出来要加进来的。

## 7. 主机控制器的初始化

好了，usb\_alloc\_dev，多么熟悉啊，这里做的就是为 Root Hub 申请了一个 struct usb\_device 结构体，并且初始化，将返回值赋给指针 rhdev。回顾这个函数可以知道，Root Hub 的 parent 指针指向了主机控制器本身。

1585 行，确定 rhdev 的 speed，UHCI 和 OHCI 都是源于曾经的 USB 1.1，而 EHCI 才是来自 USB 2.0。只有 USB 2.0 才定义了高速的设备，以前的设备只有两种速度、低速和全速。也只有 EHCI 的驱动才定义了一个 flag：HCD\_USB\_2。所以这里记录的 rhdev->speed 就是 USB\_SPEED\_FULL。

1593 行，device\_init\_wakeup，也在 Hub 驱动中见过。第 2 个参数是 1 就意味着把 Root Hub 的 Wakeup 能力打开了。正如注释里说的那样，也可以在自己的驱动里面把它关掉。

1598 行，如果 hc\_driver 中有 reset 函数，就调用它，这里的 uhci\_driver 显然是有的。回去看它的定义，可知 uhci\_init。所以这时候就要调用这个函数了，显然顾名思义，它的作用是做一些初始化。它来自 drivers/usb/host/uhci-hcd.c：

```
483 static int uhci_init(struct usb_hcd *hcd)
484 {
485     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
486     unsigned io_size = (unsigned) hcd->rsrc_len;
487     int port;
488
489     uhci->io_addr = (unsigned long) hcd->rsrc_start;
490
491     /* The UHCI spec says devices must have 2 ports, and goes on to say
492      * they may have more but gives no way to determine how many there
493      * are. However according to the UHCI spec, Bit 7 of the port
494      * status and control register is always set to 1. So we try to
495      * use this to our advantage. Another common failure mode when
496      * a nonexistent register is addressed is to return all ones, so
497      * we test for that also.
498      */
499     for (port = 0; port < (io_size - USBPORTSC1) / 2; port++) {
500         unsigned int portstatus;
501
502         portstatus = inw(uhci->io_addr + USBPORTSC1 + (port * 2));
```



```

503         if (!(portstatus & 0x0080) || portstatus == 0xffff)
504             break;
505     }
506     if (debug)
507         dev_info(uhci_dev(uhci), "detected %d ports\n", port);
508
509     /* Anything greater than 7 is weird so we'll ignore it. */
510     if (port > UHCI_RH_MAXCHILD) {
511         dev_info(uhci_dev(uhci), "port count misdetected? "
512             "forcing to 2 ports\n");
513         port = 2;
514     }
515     uhci->rh_numports = port;
516
517     /* Kick BIOS off this hardware and reset if the controller
518      * isn't already safely quiescent.
519      */
520     check_and_reset_hc(uhci);
521     return 0;
522 }

```

可恶！又出来一个新的结构体：struct uhci\_hcd，很显然，这是 UHCI 特有的。

```

371 struct uhci_hcd {
372
373     /* debugfs */
374     struct dentry *dentry;
375
376     /* Grabbed from PCI */
377     unsigned long io_addr;
378
379     struct dma_pool *qh_pool;
380     struct dma_pool *td_pool;
381
382     struct uhci_td *term_td;          /* Terminating TD, see UHCI bug */
383     struct uhci_qh *skelqh[UHCI_NUM_SKELQH]; /* Skeleton QHs */
384     struct uhci_qh *next_qh;        /* Next QH to scan */
385
386     spinlock_t lock;
387
388     dma_addr_t frame_dma_handle; /* Hardware frame list */
389     __le32 *frame;
390     void **frame_cpu;            /* CPU's frame list */
391
392     enum uhci_rh_state rh_state;
393     unsigned long auto_stop_time; /* When to AUTO_STOP */
394
395     unsigned int frame_number; /* As of last check */
396     unsigned int is_stopped;
397 #define UHCI_IS_STOPPED 9999 /* Larger than a frame # */
398     unsigned int last_iso_frame; /* Frame of last scan */
399     unsigned int cur_iso_frame; /* Frame for current scan */
400
401     unsigned int scan_in_progress:1; /* Schedule scan is running */
402     unsigned int need_rescan:1; /* Redo the schedule scan */
403     unsigned int dead:1; /* Controller has died */
404     unsigned int working_RD:1; /* Suspended root hub doesn't
405                                need to be polled */

```

```

406     unsigned int is_initialized:1;      /* Data structure is usable */
407     unsigned int fsbr_is_on:1;          /* FSBR is turned on */
408     unsigned int fsbr_is_wanted:1;      /* Does any URB want FSBR? */
409     unsigned int fsbr_expiring:1;      /* FSBR is timing out */
410
411     struct timer_list fsbr_timer;        /* For turning off FSBR */
412
413     /* Support for port suspend/resume/reset */
414     unsigned long port_c_suspend;        /* Bit-arrays of ports */
415     unsigned long resuming_ports;
416     unsigned long ports_timeout;        /* Time to stop signalling */
417
418     struct list_head idle_qh_list;       /* Where the idle QHs live */
419
420     int rh_numports;                    /* Number of root-hub ports */
421
422     wait_queue_head_t waitqh;           /* endpoint_disable waiters */
423     int num_waiting;                    /* Number of waiters */
424
425     int total_load;                     /* Sum of array values */
426     short load[MAX_PHASE];              /* Periodic allocations */
427 };

```

写代码的人永远都不会体会到读代码人的痛苦，我真的觉得如果这些变态的数据结构再多出现几个的话就不想看了。我的忍耐是有底线的。

看 uhci\_init 的 485 行，hcd\_to\_uhci，来自 drivers/usb/host/uhci-hcd.h:

```

430 static inline struct uhci_hcd *hcd_to_uhci(struct usb_hcd *hcd)
431 {
432     return (struct uhci_hcd *) (hcd->hcd_priv);
433 }
434 static inline struct usb_hcd *uhci_to_hcd(struct uhci_hcd *uhci)
435 {
436     return container_of((void *) uhci, struct usb_hcd, hcd_priv);
437 }

```

很显然，这两个函数完成的的就是 uhci\_hcd 和 usb\_hcd 之间的转换。至于 hcd->hcd\_priv 是什么？首先我们看到 struct usb\_hcd 中有一个成员 unsigned long hcd\_priv[0]。这就是传说中的零长度数组！

执行这个命令：

```
localhost:~ # info gcc "c ext" zero
```

就会知道什么是 GCC 中所谓的零长度数组。这是 GCC 对 C 的扩展。在标准 C 中我们定义数组时其长度至少为 1，而在 Linux 内核中结构体的定义里却经常看到最后一个元素定义为这种零长度数组，不占结构的空间，但它意味着这个结构体的长度充满了变数，即 sizeof(hcd\_priv)==0 的作用就相当于一个占位符。申请空间时可使用它：

```
hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
```

实际上，这就是 usb\_create\_hcd 中的一行，只不过当时没讲，现在知道，逃避不是办法，

必须面对。driver->hcd\_priv\_size 可以在 uhci\_driver 中找到，即 sizeof(struct uhci\_hcd)，所以最终 hcd->hcd\_priv 代表的就是 struct uhci\_hcd，只不过需要一个强制转换。这样就能理解 hcd\_to\_uhci 了。当然，反过来，uhci\_to\_hcd 的作用就更显而易见了。不过我倒是想友情提醒一下，现在是在讲 UHCI 主机控制器的驱动程序，那么在这个故事里，有一些结构体变量是唯一的，比如 struct uhci\_hcd 的结构体指针，即 uhci；而 struct usb\_hcd 的结构体指针即 hcd。也就是以后我们如果见到某个指针名字为 uhci 或者 hcd，那就不用再多解释了。此 uhci 即彼 uhci，此 hcd 即彼 hcd。

接下来，io\_size 和 io\_addr 的赋值都很好懂。

然后是决定这个 Root Hub 到底有几个端口的时候。端口号是从 0 开始的，UHCI 的 Root Hub 最多不能超过 8 个端口，即端口号不能超过 7。这段代码的含义注释里面说得很清楚，首先 UHCI 定义了 PORTSC 寄存器，全称为 PORT STATUS AND CONTROL REGISTER，即端口状态和控制寄存器，只要有一个端口就有一个寄存器。而每个这类寄存器都是 16 个 bits，即两个 Bytes，因此从地址安排上来说，每一个端口占两个 Bytes，而 spec 规定端口 1 的地址位于基址开始的第 10h 和 11h，端口 2 的地址向后顺推，即位于基址开始的第 12h 和 13h，依此类推，USBPORTSC1 这个宏的值是 16，USBPORTSC2 的宏值为 18，这两个宏用来标志端口的偏移量，显然 16 就是 10h，而 18 就是 12h。UHCI 定义的寄存器中，PORTSC 是最后一类寄存器，它后面没有更多的寄存器了，但是它究竟有几个 PORTSC 寄存器就不知道了，否则也就不需要判断有多少个端口了。

于是这段代码就从 USBPORTSC1 开始往后走，一直循环下去，读取这个寄存器的值，即 portstatus，按 spec 规定，这个寄存器的值中 bit7 是一个保留位，应该一直为 1。所以如果不为 1，那么就不用往下走了，说明已经没有寄存器了。另一种常见的错误是读出来都为 1，经验表明，这种情况也表示没有寄存器了。说明一下，inw 就是读 IO 端口的函数，w 就表示按 word 读。很显然这里要按 word 读，因为 PORTSC 寄存器是 16bits，一个字（word）。inw 所接的参数就是具体的 I/O 地址，即基址加偏移量。

510 行，UHCI\_RH\_MAXCHILD 就是 7，port 不能大于 7，否则出错了，于是设置 port 为 2，因为 UHCI spec 规定每个 Root Hub 最少有两个端口。

于是，uhci->rh\_numports 被用来记录 Root Hub 的端口数。

520 行，check\_and\_reset\_hc() 函数特“虚伪”，看似“超简单”其实“超复杂”。定义于 drivers/usb/host/uhci-hcd.c 中：

```
169 static void check_and_reset_hc(struct uhci_hcd *uhci)
170 {
171     if (uhci_check_and_reset_hc(to_pci_dev(uhci_dev(uhci)),
                                uhci->io_addr))
172         finish_reset(uhci);
173 }
```

看上去就两行,可这两行足以让我看了半个小时。首先第一个函数,uhci\_check\_and\_reset\_hc 来自 drivers/usb/host/pci-quirks.c,

```

91 int uhci_check_and_reset_hc(struct pci_dev *pdev, unsigned long base)
92 {
93     u16 legsup;
94     unsigned int cmd, intr;
95
96     /*
97      * When restarting a suspended controller, we expect all the
98      * settings to be the same as we left them:
99      *
100     *     PIRQ and SMI disabled, no R/W bits set in USBLEGSUP;
101     *     Controller is stopped and configured with EGSM set;
102     *     No interrupts enabled except possibly Resume Detect.
103     *
104     * If any of these conditions are violated we do a complete reset.
105     */
106     pci_read_config_word(pdev, UHCI_USBLEGSUP, &legsup);
107     if (legsup & ~(UHCI_USBLEGSUP_RO | UHCI_USBLEGSUP_RWC)) {
108         dev_dbg(&pdev->dev, "%s: legsup = 0x%04x\n",
109             __FUNCTION__, legsup);
110         goto reset_needed;
111     }
112
113     cmd = inw(base + UHCI_USBCMD);
114     if ((cmd & UHCI_USBCMD_RUN) || !(cmd & UHCI_USBCMD_CONFIGURE) ||
115         !(cmd & UHCI_USBCMD_EGSM)) {
116         dev_dbg(&pdev->dev, "%s: cmd = 0x%04x\n",
117             __FUNCTION__, cmd);
118         goto reset_needed;
119     }
120
121     intr = inw(base + UHCI_USBINTR);
122     if (intr & (~UHCI_USBINTR_RESUME)) {
123         dev_dbg(&pdev->dev, "%s: intr = 0x%04x\n",
124             __FUNCTION__, intr);
125         goto reset_needed;
126     }
127     return 0;
128
129 reset_needed:
130     dev_dbg(&pdev->dev, "Performing full reset\n");
131     uhci_reset_hc(pdev, base);
132     return 1;
133 }

```

而第二个函数 finish\_reset 来自 drivers/usb/host/uhci-hcd.c:

```

129 static void finish_reset(struct uhci_hcd *uhci)
130 {
131     int port;
132
133     /* HCRESET doesn't affect the Suspend, Reset, and Resume Detect
134     * bits in the port status and control registers.
135     * We have to clear them by hand.
136     */

```

```

137     for (port = 0; port < uhci->rh_numports; ++port)
138         outw(0, uhci->io_addr + USBPORTSC1 + (port * 2));
139
140     uhci->port_c_suspend = uhci->resuming_ports = 0;
141     uhci->rh_state = UHCI_RH_RESET;
142     uhci->is_stopped = UHCI_IS_STOPPED;
143     uhci_to_hcd(uhci)->state = HC_STATE_HALT;
144     uhci_to_hcd(uhci)->poll_rh = 0;
145
146     uhci->dead = 0;          /* Full reset resurrects the controller */
147 }

```

一起看这两个函数。`pci_read_config_word` 函数的作用正如同它的字面意思一样：读寄存器，读什么寄存器？就是那张“上坟图”呗。

在 `drivers/usb/host/quirks.c` 中有一打关于这些宏的定义：

```

20 #define UHCI_USBLEGSUP      0xc0      /* legacy support */
21 #define UHCI_USBCMD         0          /* command register */
22 #define UHCI_USBINTR        4          /* interrupt register */
23 #define UHCI_USBLEGSUP_RWC  0x8f00    /* the R/WC bits */
24 #define UHCI_USBLEGSUP_RO   0x5040    /* R/O and reserved bits */
25 #define UHCI_USBCMD_RUN     0x0001    /* RUN/STOP bit */
26 #define UHCI_USBCMD_HCRESET 0x0002    /* Host Controller reset */
27 #define UHCI_USBCMD_EGSM    0x0008    /* Global Suspend Mode */
28 #define UHCI_USBCMD_CONFIGURE 0x0040  /* Config Flag */
29 #define UHCI_USBINTR_RESUME 0x0002    /* Resume interrupt enable */

```

`UHCI_USBLEGSUP` 是一个比较特殊的寄存器，`LEGSUP` 全称为 `LEGACY SUPPORT REGISTER`，`UHCI spec` 的 5.2 节专门介绍了这个寄存器。一调用 `pci_read_config_word` 函数就是把这个寄存器的值读出来，而把结果保存在变量 `legsup` 中，接着下一行就来判断它的值。

这里首先介绍三个概念。我们注意到寄存器的每一位都有一个属性，比如 `RO`（Read Only，只读），`RW`（Read/Write，可读可写），`RWC`（Read/Write Clear）。寄存器的某位如果是 `RWC` 的属性，那么表示该位可以被读可以被写，然而，与 `RW` 不同的是，如果写了一个 1 到该位，将会把该位清为 0。倘若你写的是一个 0，则什么也不会发生，对这个世界不产生任何改变。

（`UHCI Spec` 中是这么说的：“`R/WC Read/Write Clear. A register bit with this attribute can be read and written. However, a write of a 1 clears (sets to 0) the corresponding bit and a write of a 0 has no effect.`”）

而 `UHCI_USBLEGSUP_RO` 为 `0x5040`，即 `0101 0000 0100 0000`，标志 `LEGSUP` 寄存器的 bit 6，bit 12 和 bit 14 三位为 1，这几位是只读的（bit 14 是保留位）。`UHCI_USBLEGSUP_RWC` 为 `0x8f00`，即 `1000 1111 0000 0000`，即标志着 `LEGSUP` 寄存器的 bit 8，bit 9，bit 10，bit 11 和 bit 15 为 1，这几位的属性是 `RWC` 的。这里让这两个宏“或”一下，按位去反，然后让 `legsup` 和它们相与，其效果就是判断 `LEGSUP` 寄存器的 bit 0，bit 1，bit 2，bit 3，bit 4，bit 5，bit 7 和 bit 13 是否为 1，这几位其实就是 `RW` 的。

这里的注释说，这几位任何一位为 1 则表示需要“reset”，其实这种注释是不太负责任的，

仔细看一下 UHCI spec 会发现，RW 的这些位并不是因为它们都是 RW 位就应该被 reset，而是因为 bit0~bit5，bit7，bit13 实际上都是一些 enable/disable 的开关，特别是中断相关的使能位，当我们还没有准备就绪时，我们理应把它们关掉。就相当于我新买了一个手机，而还没有号码，那我出于省电的考虑，基本上会选择把手机先关掉，等到我有号了，才会去把手机打开。而其他位都是一些状态位，它们为 0 还是为 1 只是表明不同的状态。状态位影响不大。

113 行，UHCI\_USBCMD 表征 UHCI 的命令寄存器，这也是 UHCI spec 中定义的寄存器。这个寄存器为 00h 和 01h 处，所以 UHCI\_USBCMD 的值为 0。

关于这个寄存器，需要考虑几位。首先是 bit 0，这一位被称作 RS bit，即 Run/Stop，当这一位被设置为 1，表示 HC 开始处理执行调度，调度什么？比如，传说中的 urb。显然，现在时机还未成熟，所以这一位必须设置为 0。这里代码的意思是如果它为 1，就执行 reset。UHCI\_USBCMD\_RUN 的值为 0x0001，即表征 bit 0。

另两个需要考虑的是 bit 3 和 bit 6。bit 3 被称为 EGSM，即 Enter Global Suspend Mode，这一位为 1 表示 HC 进入 Global Suspend Mode，这期间是不会有 USB 交易的。把这一位设置为 0 则是跳出这个模式，显然咱们这里的判断是如果这一位被设置为了 0，就执行 reset，否则就没有必要。因为 reset 的目的是为了清除当前存在于总线上的任何交易，让主机控制器和设备都“忘了过去，重新开始新的生活”。

而 bit 6 被称为 CF，即 Configure Flag，设置了这一位表示主机控制器当前正在被配置的过程中，显然如果主机控制器还在这个阶段就没有必要 reset 了。从逻辑上来说，关于这个寄存器的判断，我们的理念是，如果 HC 是停止挂起的，并且它还是配置的，就没有必要“reset”。

接下来，121 行，读另一个寄存器，UHCI\_USBINTR，值为 4。UHCI spec 中定义了一个中断使能寄存器。其 I/O 地址位于 04h 和 05h。很显然一开始我们得关中断。关于这个寄存器的描述如表 3.7.1 所示。

表 3.7.1 UHCI\_USBINTR 寄存器

Bit	描述
15:4	保留
3	Short Packet Interrupt（短包中断）Enable. 1=Enable. 0=Disable.
2	Interrupt On Complete（完成时中断，IOC）Enable. 1=Enable. 0=Disable.
1	Resume Interrupt（唤醒中断）Enable. 1=Enable. 0=Disable.
0	Timeout/CRC Interrupt（超时/CRC）Enable. 1=Enable. 0=Disable.

谢天谢地，bit 15 到 bit 4 是保留位，并且默认应该是 0，所以无需理睬。而剩下几位在现阶段应该要 disable 掉，或者说应该设置为 0。唯有 bit 1 是个例外，正如 uhci\_check\_and\_reset\_hc 函数前的注释里说的一样，调用这个函数有两种可能的上下文：一种是主机控制器刚刚被发现时，这是一次性的工作；另一种是电源管理中的“resume”之时。虽然此时此刻调用这个函数是处于第一种上下文，但显然第二种情景发生的频率更高，可能性更大。

对于“resume”的情况，显然这个唤醒中断使能寄存器必须被 enable。（这里也能解释为何不应该清掉 LEGSUP 寄存器里面的状态位，还应该尽量保持它们。因为如果从“suspend”回到“resume”，当然希望之前的状态得到保留，否则状态改变了那不就乱了么？那也就不叫恢复了。）

最终，127 行，如果前面的三条 goto 语句都没有执行，那么说明并不需要执行“reset”，这里就直接返回了，返回值为 0。反之如果前面任何一条 goto 语句执行了，那么就往下走，执行 uhci\_reset\_hc，然后返回 1。

函数 uhci\_reset\_hc 也来自 drivers/usb/host/pci-quirks.c:

```
59 void uhci_reset_hc(struct pci_dev *pdev, unsigned long base)
60 {
61     /* Turn off PIRQ enable and SMI enable. (This also turns off the
62      * BIOS's USB Legacy Support.) Turn off all the R/WC bits too.
63      */
64     pci_write_config_word(pdev, UHCI_USBLEGSUP, UHCI_USBLEGSUP_RWC);
65
66     /* Reset the HC - this will force us to get a
67      * new notification of any already connected
68      * ports due to the virtual disconnect that it
69      * implies.
70      */
71     outw(UHCI_USBCMD_HCRESET, base + UHCI_USBCMD);
72     mb();
73     udelay(5);
74     if (inw(base + UHCI_USBCMD) & UHCI_USBCMD_HCRESET)
75         dev_warn(&pdev->dev, "HCRESET not completed yet!\n");
76
77     /* Just to be safe, disable interrupt requests and
78      * make sure the controller is stopped.
79      */
80     outw(0, base + UHCI_USBINTR);
81     outw(0, base + UHCI_USBCMD);
82 }
```

这个函数其实就是一堆寄存器操作：读寄存器或者写寄存器。这种代码完全就是纸老虎，看上去挺恐怖，一堆的宏啊，寄存器啊，其实这些东西对我们完全就是小菜一碟。

首先 64 行，pci\_write\_config\_word 就是写寄存器，写的还是 UHCI\_USBLEGSUP 寄存器，即 LEGSUP 寄存器，写入 UHCI\_USBLEGSUP\_RWC，根据对 RWC 的解释，这样做的后果就是让这几位都清零。凡是 RWC 的 bit 其实都是在传达某种事件的发生，而清零往往代表的是认可这件事情。

然后 71 行，outw 的作用就是写端口地址，这里写的是 UHCI\_USBCMD，即写命令寄存器，而 UHCI\_USBCMD\_HCRESET 表示什么意思呢？UHCI spec 中是这样说的：“Host Controller Reset (HCRESET). When this bit is set, the Host Controller module resets its internal timers, counters, state machines, etc. to their initial value. Any transaction currently in progress on USB is immediately

terminated. This bit is reset by the Host Controller when the reset process is complete.”

显然，这就是真正的执行硬件上的“reset”。把计时器、计数器和状态机全都给复位。当然这件事情是需要一段时间的，所以这里调用 `udelay` 来延时 5 ms。最后再读这一位，因为正如上面这段英文里所说的那样，当“reset”完成了之后，这一位会被硬件“reset”，即这一位应该为 0。

最后 80 行和 81 行，写寄存器，把 0 写入中断使能寄存器和命令寄存器，这就是彻底的“reset”，关掉中断请求，并且停止 HC。

终于，我们结束了 `uhci_check_and_reset_hc`，如果执行了 `reset`，返回值应该为 1。则将执行 `finish_reset` 函数。这个函数的代码前面已经贴出来了，因为它只是做一些简单的赋值。唯一有一行写寄存器的循环操作，其含义又在注释里写得很明白。至于这些赋值究竟意味着什么，之后遇到了再说。

于是又跳出了 `check_and_reset_hc(uhci)`，回到了 `uhci_init`，不过幸运的是，这个函数也该结束了，返回值为 0，就来个三级跳，回到了 `usb_add_hcd`。

## 8. 有一种资源，叫中断

结束了 `uhci_init` 回到亲爱的 `usb_add_hcd` 之后，1604 行到 1606 行是调试语句，飘过。

有一种液体叫眼泪，曾经以为，闭上眼睛，眼泪就不会流出来了。的确，眼泪流回了心里。

有一种资源叫中断，曾经以为，关掉中断，USB 主机就不会工作了。的确，USB 主机没有中断基本就“挂了”……

1609 行，中断，判断 `driver` 有没有一个 `irq` 函数，如果没有，就简单地记录 `hcd->irq` 为 -1（`hcd->irq` 就是用来记录传说中的中断号的）。如果有，那就有事情要做了。显然 `uhci_driver` 的赋值就是 `uhci_irq`。当然，现在不用执行它，只要为它做点事情。最重要的是 `request_irq` 这个函数。我们先来查看它直观的效果。

加载 `uhci-hcd` 之前：

```
localhost:~ # cat /proc/interrupts
CPU0
0: 29906 IO-APIC-edge timer
1: 10 IO-APIC-edge i8042
8: 2 IO-APIC-edge rtc
9: 3 IO-APIC-fastEOI acpi
12: 115 IO-APIC-edge i8042
```



```
14:      6800   IO-APIC-edge    ide0
16:      780   IO-APIC-fasteoi  eth0
NMI:      0
LOC:     1403
ERR:      0
MIS:      0
```

加载 uhci-hcd 模块:

```
localhost:~ # modprobe usbcore
localhost:~ # modprobe uhci-hcd
```

加载 uhci-hcd 模块之后:

```
localhost:~ # cat /proc/interrupts
CPU0
0:      32625   IO-APIC-edge    timer
1:       10   IO-APIC-edge    i8042
8:       2   IO-APIC-edge    rtc
9:       3   IO-APIC-fasteoi  acpi
12:      115   IO-APIC-edge    i8042
14:     6915   IO-APIC-edge    ide0
16:      870   IO-APIC-fasteoi  eth0, uhci_hcd:usb1
17:       0   IO-APIC-fasteoi  uhci_hcd:usb4
18:       0   IO-APIC-fasteoi  uhci_hcd:usb2
19:       0   IO-APIC-fasteoi  uhci_hcd:usb3
NMI:      0
LOC:     1403
ERR:      0
MIS:      0
```

众所周知，`/proc/interrupts` 列出了计算机中中断资源的使用情况。这其中 `uhci_hcd:usb1/usb2/usb3/usb4` 这几个字符串就是 `request_irq` 中的倒数第二个参数，即实参 `hcd->irq_descr`，而这个字符串的赋值就是 1610 行那个 `snprintf` 语句的职责。`hcd->driver->description` 就是“uhci-hcd”，`hcd->self.busnum` 就是 1, 2, 3, 4。因为这台机器一共 4 个 usb 主机控制器，它们的编号分别是 1, 2, 3, 4。

`request_irq` 的具体作用是请求中断资源，更准确地说是安装中断处理函数或者叫中断句柄 (interrupt handler)。

这个函数的第 1 个参数就是中断号，`irqnum` 是一路传下来的，即最初的那个 `dev->irq`。

第 2 个参数就是中断句柄。这里传递的是 `usb_hcd_irq` 函数，它将会在响应中断时被调用。

第 3 个参数，`irqflags`，在 `probe` 中调用 `usb_add_hcd` 时，传递的第三个参数是 `IRQF_SHARED`，这个参数也被传递给了 `request_irq`，所以这里的 `irqflags` 为 `IRQF_SHARED`，这表示该中断可以被多个设备共享。这也是为什么 `uhci-hcd:usb1` 和网卡驱动 `eth0` 用的是同一个中断号。之所以要共享，是因为在操作系统中，也要节约资源，尽量和别的设备共享中断号。

第 4 个参数就是字符串。第 5 个参数主要是用来标志使用中断的设备，它是一个指针，这

个参数只是用来区分不同的设备，可以把它设置为 NULL，不用它。但更多的情况是它被设置为指向驱动程序私有的数据。传递的是 hcd 本身，这样在 usb\_hcd\_irq 函数中就可以使用它，因为事实上在 usb\_hcd\_irq 中把它当成一个参数来用。

这样 request\_irq 的作用就明白了，以后释放中断资源时我们只要调用 free\_irq 函数即可。这个函数将会在 usb\_remove\_hcd 时被调用。

最后 1618 行，也把 irqnum 记录在了 hcd->irq 中。

再强调一下，usb\_hcd\_irq 是这个故事中很重要的角色，它将在未来主机控制器需要中断时被调用。希望不要把她忘怀。作为一个中断函数，如果不是它以后有一定的利用价值，咱们现在完全没有必要为它注册，这非常符合常理！

## 9. 一个函数引发的故事（一）

接下来，1632 行，下一个函数，driver->start 被调用。对于 uhci\_driver，其 start 指针指向的是 uhci\_start 函数，经过了“人间大炮一级准备、二级准备”之后，这个函数基本上就算介于三级准备和发射之间了。这个函数算是整个故事中最重要的一個函数，理解它是理解整个 uhci 的关键，它来自文件 drivers/usb/host/uhci-hcd.c:

```

556 static int uhci_start(struct usb_hcd *hcd)
557 {
558     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
559     int retval = -EBUSY;
560     int i;
561     struct dentry *dentry;
562
563     hcd->uses_new_polling = 1;
564
565     spin_lock_init(&uhci->lock);
566     setup_timer(&uhci->fsbr_timer, uhci_fsbr_timeout,
567                (unsigned long) uhci);
568     INIT_LIST_HEAD(&uhci->idle_qh_list);
569     init_waitqueue_head(&uhci->waitqh);
570
571     if (DEBUG_CONFIGURED) {
572         dentry = debugfs_create_file(hcd->self.bus_name,
573                                     S_IFREG|S_IRUGO|S_IWUSR, uhci_debugfs_root,
574                                     uhci, &uhci_debug_operations);
575         if (!dentry) {
576             dev_err(uhci_dev(uhci), "couldn't create uhci "
577                    "debugfs entry\n");
578             retval = -ENOMEM;
579             goto err_create_debug_entry;
580         }
581         uhci->dentry = dentry;

```

```

582     }
583
584     uhci->frame = dma_alloc_coherent(uhci_dev(uhci),
585                                     UHCI_NUMFRAMES * sizeof(*uhci->frame),
586                                     &uhci->frame_dma_handle, 0);
587     if (!uhci->frame) {
588         dev_err(uhci_dev(uhci), "unable to allocate "
589                 "consistent memory for frame list\n");
590         goto err_alloc_frame;
591     }
592     me mset(uhci->frame, 0, UHCI_NUMFRAMES * sizeof(*uhci->frame));
593
594     uhci->frame_cpu = kcalloc(UHCI_NUMFRAMES, sizeof(*uhci->frame_cpu),
595                             GFP_KERNEL);
596     if (!uhci->frame_cpu) {
597         dev_err(uhci_dev(uhci), "unable to allocate "
598                 "memory for frame pointers\n");
599         goto err_alloc_frame_cpu;
600     }
601
602     uhci->td_pool = dma_pool_create("uhci_td", uhci_dev(uhci),
603                                     sizeof(struct uhci_td), 16, 0);
604     if (!uhci->td_pool) {
605         dev_err(uhci_dev(uhci), "unable to create td dma_pool\n");
606         goto err_create_td_pool;
607     }
608
609     uhci->qh_pool = dma_pool_create("uhci_qh", uhci_dev(uhci),
610                                     sizeof(struct uhci_qh), 16, 0);
611     if (!uhci->qh_pool) {
612         dev_err(uhci_dev(uhci), "unable to create qh dma_pool\n");
613         goto err_create_qh_pool;
614     }
615
616     uhci->term_td = uhci_alloc_td(uhci);
617     if (!uhci->term_td) {
618         dev_err(uhci_dev(uhci), "unable to allocate terminating TD\n");
619         goto err_alloc_term_td;
620     }
621
622     for (i = 0; i < UHCI_NUM_SKELQH; i++) {
623         uhci->skelqh[i] = uhci_alloc_qh(uhci, NULL, NULL);
624         if (!uhci->skelqh[i]) {
625             dev_err(uhci_dev(uhci), "unable to allocate QH\n");
626             goto err_alloc_skelqh;
627         }
628     }
629
630     /*
631     * 8 Interrupt queues; link all higher int queues to int1 = async
632     */
633     for (i = SKEL_ISO + 1; i < SKEL_ASYNC; ++i)
634         uhci->skelqh[i]->link = LINK_TO_QH(uhci->skel_async_qh);
635     uhci->skel_async_qh->link = UHCI_PTR_TERM;
636     uhci->skel_term_qh->link = LINK_TO_QH(uhci->skel_term_qh);
637
638     /* This dummy TD is to work around a bug in Intel PIIX controllers*/
639     uhci_fill_td(uhci->term_td, 0, uhci_explen(0) |
640                 (0x7f << TD_TOKEN_DEVADDR_SHIFT) | USB_PID_IN, 0);

```

```

641     uhci->term_td->link = UHCI_PTR_TERM;
642     uhci->skel_async_gh->element = uhci->skel_term_gh->element =
643         LINK_TO_TD(uhci->term_td);
644
645     /*
646     * Fill the frame list: make all entries point to the proper
647     * interrupt queue.
648     */
649     for (i = 0; i < UHCI_NUMFRAMES; i++) {
650
651         /* Only place we don't use the frame list routines */
652         uhci->frame[i] = uhci_frame_skel_link(uhci, i);
653     }
654
655     /*
656     * Some architectures require a full mb() to enforce completion of
657     * the memory writes above before the I/O transfers in configure_hc().
658     */
659     mb();
660
661     configure_hc(uhci);
662     uhci->is_initialized = 1;
663     start_rh(uhci);
664     return 0;
665
666 /*
667  * error exits:
668  */
669 err_alloc_skelqh:
670     for (i = 0; i < UHCI_NUM_SKELOH; i++) {
671         if (uhci->skelqh[i])
672             uhci_free_gh(uhci, uhci->skelqh[i]);
673     }
674
675     uhci_free_td(uhci, uhci->term_td);
676
677 err_alloc_term_td:
678     dma_pool_destroy(uhci->qh_pool);
679
680 err_create_qh_pool:
681     dma_pool_destroy(uhci->td_pool);
682
683 err_create_td_pool:
684     kfree(uhci->frame_cpu);
685
686 err_alloc_frame_cpu:
687     dma_free_coherent(uhci_dev(uhci),
688         UHCI_NUMFRAMES * sizeof(*uhci->frame),
689         uhci->frame, uhci->frame_dma_handle);
690
691 err_alloc_frame:
692     debugfs_remove(uhci->dentry);
693
694 err_create_debug_entry:
695     return retval;
696 }

```

这个函数简直就是一个大杂烩，所有“变态”的代码全都集中在这一个函数中了。天下没

有轻松的成功，成功要付代价。在这个浮躁的社会中，也许很难再有人能够静下心来来看代码了。都说现在的程序员是做一天程序撞一天钟，我们这些读程序的又何尝不是这种心态呢？

面对这越来越枯燥的代码，我想我们不能再像过去那样分析了。记得一位朋友曾经教育过我，读懂 Linux 内核代码和读懂女人的心一样，不是不可能，只是需要你多下点工夫，多用点时间，多多沟通，多多了解。这套理论我觉得很有道理，所以我想从现在开始我决定多用点时间多下点工夫来读代码，要和代码多多沟通才能对它有多多了解。所以我决定用出我的杀手锏 KDB。也许你不熟悉 KDB，没有关系，我只通过 KDB 来展示一些函数调用关系。我主要会使用 KDB 的 `bp` 命令来设置断点，用 `bt` 命令来显示函数调用堆栈。很显然，了解了函数的调用关系对读懂代码是很有帮助的。

首先在加载 `uhci-hcd` 时设置断点 `uhci_start`。于是会在 `uhci_start` 被调用时进入 KDB，用 `bt` 命令看一下堆栈的 `traceback`。

```
kdb>bt
Stack traceback for pid 3498
0xdd5ac550 3498 3345 1 0 R 0xdd5ac720 *modprobe
esp      eip      Function (args)
0xd4a89d40 0xc0110000 lapic_resume+0x185
0xd4a89d48 0xe0226e41 [uhci_hcd]uhci_start
0xd4a89d54 0xe0297132 [usbcore]usb_add_hcd+0x3fb
0xd4a89da0 0xe02a0a9b [usbcore]usb_hcd_pci_probe+0x263
```

其实堆栈中还有更多的函数，篇幅原因，与此没有太多关系的函数就不列出来了。但是至少从这个 `traceback` 中可以很清楚地知道目前处境，在 `uhci_start` 中，而调用它的函数是 `usb_add_hcd`，后者则是被 `usb_hcd_pci_probe` 函数调用，而 `usb_hcd_pci_probe` 函数正是我们故事的真正开始处。

但单单是 KDB 还不足以显示我们的决心。有人说，女人如画，不同的画中有不同的风景，代码也是如此，左看右看，上看下看，角度不同风景各异。对于 `uhci-hcd` 这样“变态”的模块，用常规的方法恐怕是很难看明白了，我们必须引入一种新的方法——图解法。从 `uhci_start` 函数开始，我们将接触到一堆乱七八糟的庞大而复杂的数据结构，这些数据结构的关系如果不能理解，那么我们很难读懂这代码。所以决定把 `uhci_start` 作为实验，通过图解法把众多复杂的结构众多的链表之间的关系给描绘出来。

---

## 10. 一个函数引发的故事（二）

571 行之前全是些初始化的代码，用到了再回来看。

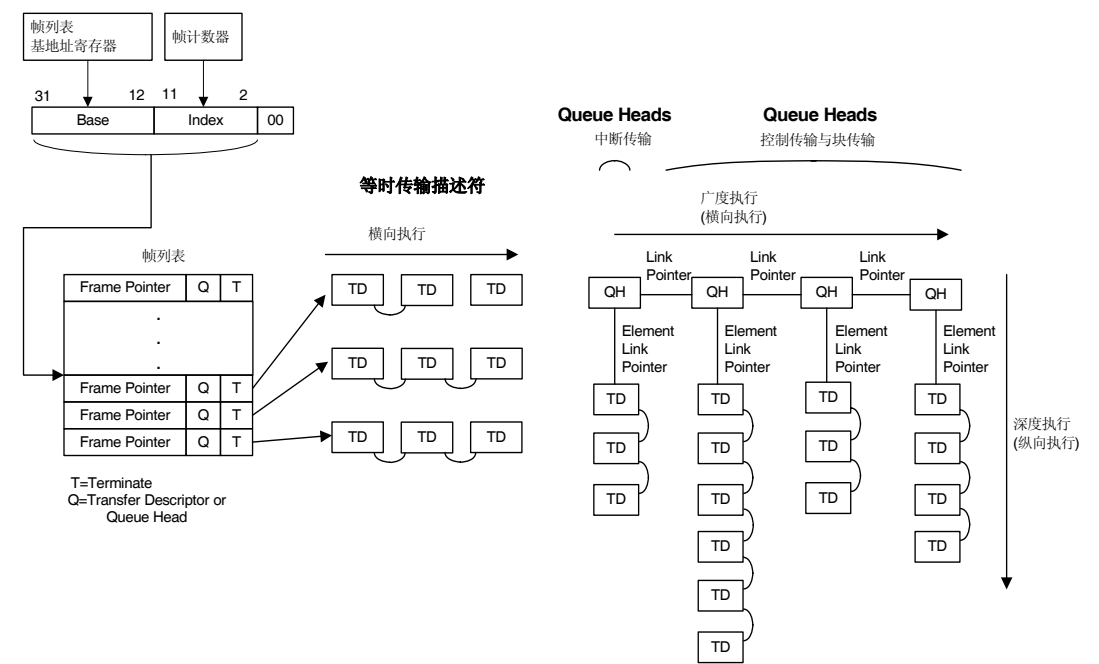
571 行到 582 行，上次我们看到 `DEBUG_CONFIGURED` 是在 UHCI 的初始化代码中，即

uhci\_hcd\_init 函数中，这是一个编译开关，打开开关就是 1，不打开就是 0，这里假设打开。因为有必要多了解一下 debugfs，毕竟当初已经接触过 debugfs 了。而且当时已经看到函数 debugfs\_create\_dir 在 /sys/kernel/debug 下面建立了一个 uhci 的目录，而现在看到的这个 debugfs\_create\_file 很自然，就是在 /sys/kernel/debug/uhci 下面建立文件，比如：

```
localhost:~ # ls -l /sys/kernel/debug/uhci/
total 0
-rw-r--r-- 1 root root 0 Oct 8 13:18 0000:00:1d.0
-rw-r--r-- 1 root root 0 Oct 8 13:18 0000:00:1d.1
-rw-r--r-- 1 root root 0 Oct 8 13:18 0000:00:1d.2
-rw-r--r-- 1 root root 0 Oct 8 13:18 0000:00:1d.3
```

可以看见，在这个目录下面针对每个 UHCI 主机控制器各建立了一个文件，文件名就是该设备的 PCI 名，即域名+总线名+插槽号+功能号。

接下来从 584 行到 628 行，全都是内存申请相关的，包括可恶的 DMA。不过这些函数已经不再陌生，dma\_alloc\_coherent, dma\_pool\_create 都已经讲过，但要说清楚这些实际的代码，则必须借助一张来自 UHCI spec 的经典图，如图 3.9.1 所示。



有了上面这张图，你就可以运筹帷幄之内决胜千里之外。这张图对于我们研究 UHCI 的意义，就好比 1993 年那部何家劲版的少年张三丰中那张藏宝图，所有人都想得到它，因为得到了它就意味着得到一切。而对于所有写 UHCI 主机驱动的人来说，他们对于这幅图的共同心声是：输了你，赢了世界又如何？

之所以这幅图如此重要，是因为 UHCI 主机控制器的原理完全集中在这幅图中。

主机控制器最重要的一个职责是调度。那么它如何调度呢？首先你得把帧列表(Frame List)的起始地址告诉它，由这个 List 将会牵出许多的 TD 和 QH 来。

首先 Frame List 是一个数组，最多 1024 个元素。每一个元素代表一个 Frame，时间上来说就是 1 ms。而和每一个 Frame 相联系的是“transaction”，即交易。比如说一次传输，就可以算作一笔交易。而 TD 和 QH 就是用来表征这些交易的。Frame List 的每一个元素会包含指针指向 TD 或者 QH，实际上 Frame List 的每一个元素被称作一个 Frame List Pointer，它一共有 32 个 bit，其中 bit31 到 bit4 则表示 TD 或者 QH 的地址，bit1 则表示指向的到底是 QH 还是 TD。

而从硬件上来说，访问这个 Frame List 的方法是使用一个基址寄存器和一个计数器，即图中我们看到的那个帧列表基址寄存器 (Frame List Base Address Register) 和帧计数器 (Frame Counter)。下面的图 3.9.2 也许更能说明它们的作用。

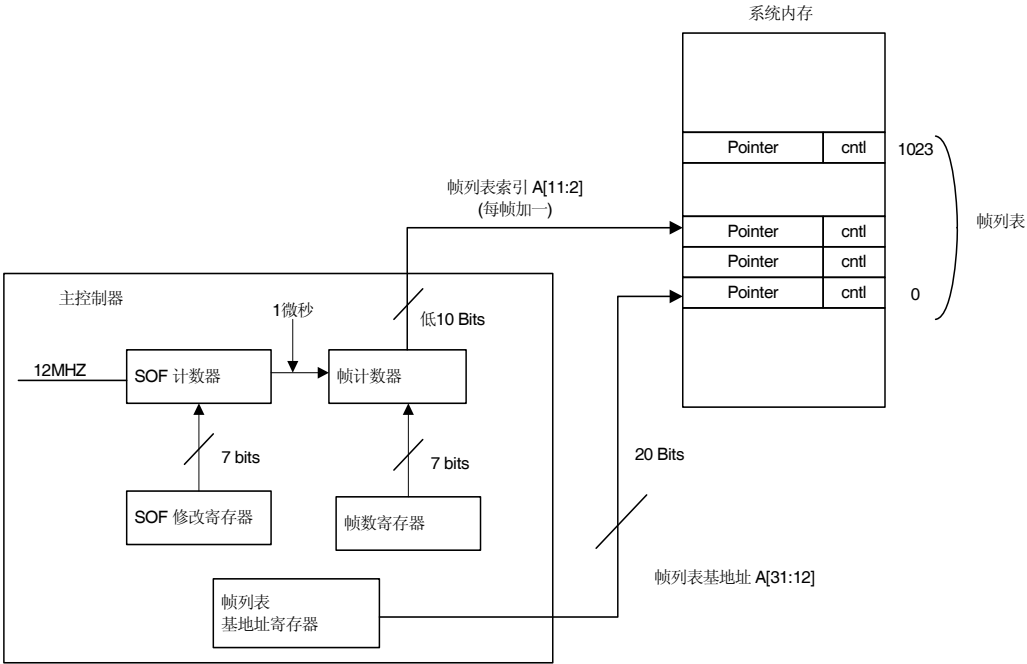


图 3.9.2 帧列表基址寄存器和帧计数器

实际上在主机控制器中有一个时钟，用我们电子专业的术语来说就是主机控制器内部肯定有一个晶体振荡器，从而实现一个周期为 1 msec 的信号，于是帧列表基址寄存器和帧计数器就去遍历这张 Frame List，它们在这张表里一毫秒前进一格，走到 1023 了就又再次归零。

那么驱动程序的责任是什么呢？为图 3.9.1 所示的那张调度图准备好相应的数据结构。填充 Frame List Pointer 建立并填充 TD 和 QH。那么到底什么是 TD 和 QH 呢？

TD 实际上描述的就是最终要在 USB 总线上传输的数据包，学名为 Transaction Descriptor，是主机控制器的基本执行单位。UHCI spec 定义了两类 TD：ISO TD 和 non-ISO TD。即等时 TD 和非等时 TD。我们知道 USB 一共四种传输方式：中断、批量、控制和等时。这其中等时传输的 TD 虽然从数据结构的格式来说是一样的，但是作用不一样。从调度图来看，等时的 TD 也是专门被列出来了。主机控制器驱动程序负责填充 TD，主机控制器将会去获取 TD，然后执行相应的数据传输。

QH 就是队列头（Queue Head）。从这张调度图里也能看到，QH 实际上把各个非等时 TD 给连接了起来，组织成若干队列。

从图中我们看到一个现象，对主机控制器来说，4 种传输方式是有优先级的区别的，等时传输总是最先被满足，最先被执行，中断传输，再控制传输和批量传输。等时传输和中断传输都叫做周期传输，或者说定期传输。

再强调一下，驱动程序的任务就是填充这张图，硬件的作用则按照这张图去执行，这种分工是很明确的。

OK，现在让我们结合代码和结构体定义来查看。

首先 584 行，使用 `dma_alloc_coherent` 申请内存，赋给 `uhci->frame`，而与此同时建立了 DMA 映射。`frame_dma_handle` 和 `frame` 是以后从软件方面来指代这个 Frame List 的，而 `frame_dma_handle` 因为是物理地址，要让它和真正的硬件联系起来，稍后在一个叫做 `configure_hc` 的函数中你会看到，我们会把它写入 Frame List 的基址寄存器。这样以后操作 `uhci->frame` 就等于真正的操作硬件了。这更意味着以后只要把申请的 TD、QH 和 `uhci->frame` 联系起来就可以了。这里我们也注意到，`UHCI_NUMFRAMES` 就是一个宏，它的值正是 1024。到目前为止，一切看起来都那么和谐。

而 594 行这个 `frame_cpu` 则是和 `frame` 相对应的，它是一个纯粹软件意义上的 Frame List。即 Frame 身上承担着硬件的使命，而 `frame_cpu` 则属于从软件角度来说记录这张 Frame List。

609 行，这两个 `dma_poll_create` 的作用就是创建内存池，为 TD 和 QH 申请内存带来方便。

616 行调用的这个 `uhci_alloc_td` 以及 623 行调用的 `uhci_alloc_qh` 则均来自 `drivers/usb/host/uhci-q.c`，先看 `uhci_alloc_td`，及其搭档 `uhci_free_td`。

```
106 static struct uhci_td *uhci_alloc_td(struct uhci_hcd *uhci)
107 {
108     dma_addr_t dma_handle;
109     struct uhci_td *td;
110
111     td = dma_pool_alloc(uhci->td_pool, GFP_ATOMIC, &dma_handle);
112     if (!td)
113         return NULL;
114 }
```



```

115     td->dma_handle = dma_handle;
116     td->frame = -1;
117
118     INIT_LIST_HEAD(&td->list);
119     INIT_LIST_HEAD(&td->fl_list);
120
121     return td;
122 }
123
124 static void uhci_free_td(struct uhci_hcd *uhci, struct uhci_td *td)
125 {
126     if (!list_empty(&td->list)) {
127         dev_warn(uhci_dev(uhci), "td %p still in list!\n", td);
128         WARN_ON(1);
129     }
130     if (!list_empty(&td->fl_list)) {
131         dev_warn(uhci_dev(uhci), "td %p still in fl_list!\n", td);
132         WARN_ON(1);
133     }
134
135     dma_pool_free(uhci->td_pool, td, td->dma_handle);
136 }

```

这意思很明白。再来看后者，uhci\_alloc\_qh 及其搭档 uhci\_free\_qh。

```

247 static struct uhci_qh *uhci_alloc_qh(struct uhci_hcd *uhci,
248                                     struct usb_device *udev, struct usb_host_endpoint *hep)
249 {
250     dma_addr_t dma_handle;
251     struct uhci_qh *qh;
252
253     qh = dma_pool_alloc(uhci->qh_pool, GFP_ATOMIC, &dma_handle);
254     if (!qh)
255         return NULL;
256
257     memset(qh, 0, sizeof(*qh));
258     qh->dma_handle = dma_handle;
259
260     qh->element = UHCI_PTR_TERM;
261     qh->link = UHCI_PTR_TERM;
262
263     INIT_LIST_HEAD(&qh->queue);
264     INIT_LIST_HEAD(&qh->node);
265
266     if (udev) { /* Normal QH */
267         qh->type = hep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK;
268         if (qh->type != USB_ENDPOINT_XFER_ISOC) {
269             qh->dummy_td = uhci_alloc_td(uhci);
270             if (!qh->dummy_td) {
271                 dma_pool_free(uhci->qh_pool, qh, dma_handle);
272                 return NULL;
273             }
274         }
275         qh->state = QH_STATE_IDLE;
276         qh->hep = hep;
277         qh->udev = udev;
278         hep->hcpriv = qh;
279     }

```

```

280         if (qh->type == USB_ENDPOINT_XFER_INT ||
281             qh->type == USB_ENDPOINT_XFER_ISOC)
282             qh->load = usb_calc_bus_time(udev->speed,
283                                         usb_endpoint_dir_in(&hep->desc),
284                                         qh->type == USB_ENDPOINT_XFER_ISOC,
285                                         le16_to_cpu(hep->desc.wMaxPacketSize))
286                               / 1000 + 1;
287
288     } else {                /* Skeleton QH */
289         qh->state = QH_STATE_ACTIVE;
290         qh->type = -1;
291     }
292     return qh;
293 }
294
295 static void uhci_free_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
296 {
297     WARN_ON(qh->state != QH_STATE_IDLE && qh->udev);
298     if (!list_empty(&qh->queue)) {
299         dev_warn(uhci_dev(uhci), "qh %p list not empty!\n", qh);
300         WARN_ON(1);
301     }
302
303     list_del(&qh->node);
304     if (qh->udev) {
305         qh->hep->hcpriv = NULL;
306         if (qh->dummy_td)
307             uhci_free_td(uhci, qh->dummy_td);
308     }
309     dma_pool_free(uhci->qh_pool, qh, qh->dma_handle);
310 }

```

这个就明显复杂一些了。uhci\_alloc\_qh 的执行有两条路径：一种是 udev 有所指，一种是 udev 为空。我们这里传进来的是 NULL，所以暂时可以不看另一种路径，到时候需要看时再去。这么一来就意味着此时此刻不需要看 266 到 287 这些行了。而这意味着能看到的只是一些简单的赋值操作而已。但是我们必须理解为什么这里有两种情况，因为这正是 uhci-hcd 这个驱动的精妙之处。

为了堆砌出那幅调度图，有一个很简单的方法，即每次有传输任务，建立一个或者几个 TD，把 urb 转换成 urb，然后把 TD 挂入 Frame List Pointer，不就可以了么？朋友，如果生活真的是这么简单，如果世界真的是这么单纯，那么也许现在的我依然是一张洁白的宣纸，绝不是现在这张被上海的梅雨湿润过的废纸。

## 11. 一个函数引发的故事（三）

从调度图可以看出，等时传输不需要 QH，只要把几个 TD 连接起来，让 Frame List Pointer 指向第一个 TD 就可以了。换言之，需要为等时传输准备一个队列，然后每一个 Frame 都让 Frame

List Pointer 指向队列的头部。

那么应该如何操作中断传输呢？实际上为中断传输建立了 8 个队列，不同的队列代表了不同的周期，这 8 个队列分别代表的是 1 ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms, 64 ms 和 128 ms。USB spec 里规定，对于全速/低速设备来说，其渴望周期撑死也不能超过 255 ms。这 8 个队列的队列头就叫做 Skeleton QH，而对于正儿八经的传输来说，我们需要另外专门的建立 QH，并往该 QH 中连接上相关的 TD，这类 QH 就是这里所称的 Normal QH。有人偏向于把往 QH 上连接 TD 称之为“为 QH 装备上 TD”。而这几个 Skeleton QH 是不会被装备任何 TD 的，要让别的中断 QH 知道自己该放置在何处。不过 Skeleton QH 并非只是为中断传输准备的，实际上，我们准备了 11 个 Skeleton QH，除了中断传输的 8 个以外，还有一个为等时传输准备的 QH，一个为表征“大部队结束”的 QH，一个为处理 unlink 而设计的 QH。这三个都有点特殊，我们遇到了再讲。

回到代码中来，从 uhci\_start 函数的角度来看，我们注意到 uhci\_alloc\_qh 返回值是 qh，而 qh 是一个 struct uhci\_qh 结构体变量，而刚才 uhci\_alloc\_td 函数的返回值 td，是一个 struct uhci\_td 结构体变量。TD 和 QH 这两个概念说起来轻松，可是化成代码来表示的这两个结构体绝对不是省油的灯。先看 struct uhci\_td，来自 drivers/usb/host/uhci-hcd.h 中：

```
242 struct uhci_td {
243     /* Hardware fields */
244     __le32 link;
245     __le32 status;
246     __le32 token;
247     __le32 buffer;
248
249     /* Software fields */
250     dma_addr_t dma_handle;
251
252     struct list_head list;
253
254     int frame; /* for iso: what frame? */
255     struct list_head fl_list;
256 } __attribute__((aligned(16)));
```

再看 struct uhci\_qh，依然来自 drivers/usb/host/uhci-hcd.h：

```
126 struct uhci_qh {
127     /* Hardware fields */
128     __le32 link; /* Next QH in the schedule */
129     __le32 element; /* Queue element (TD) pointer */
130
131     /* Software fields */
132     dma_addr_t dma_handle;
133
134     struct list_head node; /* Node in the list of QHs */
135     struct usb_host_endpoint *hep; /* Endpoint information */
136     struct usb_device *udev;
137     struct list_head queue; /* Queue of urbps for this QH */
138     struct uhci_td *dummy_td; /* Dummy TD to end the queue */
139     struct uhci_td *post_td; /* Last TD completed */
140 }
```

```

140
141     struct usb_iso_packet_descriptor *iso_packet_desc;
142                                     /* Next urb->iso_frame_desc entry */
143     unsigned long advance_jiffies; /* Time of last queue advance */
144     unsigned int unlink_frame;     /* When the QH was unlinked */
145     unsigned int period;           /* For Interrupt and Isochronous QHs */
146     short phase;                   /* Between 0 and period-1 */
147     short load;                    /* Periodic time requirement, in us */
148     unsigned int iso_frame;         /* Frame # for iso_packet_desc */
149     int iso_status;                 /* Status for Isochronous URBs */
150
151     int state;                      /* QH_STATE_xxx; see above */
152     int type;                       /* Queue type (control, bulk, etc) */
153     int skel;                       /* Skeleton queue number */
154
155     unsigned int initial_toggle:1; /* Endpoint's current toggle value */
156     unsigned int needs_fixup:1;    /* Must fix the TD toggle values */
157     unsigned int is_stopped:1;     /* Queue was stopped by error/unlink */
158     unsigned int wait_expired:1;   /* QH_WAIT_TIMEOUT has expired */
159     unsigned int bandwidth_reserved:1; /* Periodic bandwidth has
160                                         * been allocated */
161 } __attribute__((aligned(16)));

```

某种意义上来说，struct usb\_hcd，struct uhci\_hcd 这些结构体和 struct uhci\_td，struct uhci\_qh 之间的关系就好比宏观经济学与微观经济学的关系。它们都是为了描述主机控制器，只是分别从宏观角度和微观角度。从另一个角度来说，这些宏观的数据结构实际上是软件意义上的，而这些微观的数据结构倒是和硬件有着对应关系。从硬件上来说，Frame List, Isochronous Transfer Descriptors（简称 TD），Queue Heads（简称 QH），以及 queued Transfer Descriptors（也简称 TD）都是 UHCI spec 定义的数据结构。

先看 struct uhci\_td 结构体。uhci\_alloc\_td 函数定义了两个局部变量：dma\_handle 和 td，其中 dma\_handle 传递给了 dma\_pool\_alloc 函数，于是知道它记录的是 td 的 DMA 地址。td 内部有一个成员 dma\_addr\_t dma\_handle，它被赋值为 dma\_handle。td 内部另一个成员 int frame，用来表征这个 td 所对应的 frame，目前初始化 frame 为-1。另外，td 还有两个成员，struct list\_head list 和 struct list\_head fl\_list。这是两个队列。uhci\_alloc\_td 函数中用 INIT\_LIST\_HEAD 把它们俩进行了初始化。而反过来 uhci\_free\_td 函数的工作就是反其道而行之，调用 dma\_pool\_free 函数去释放这个内存。在释放之前检查了一下这两个队列是否为空，如果不为空会先提出警告。

再来看 struct uhci\_qh 结构体，在 uhci\_alloc\_qh 函数中，首先也是定义两个局部变量：qh 和 dma\_handle。使用的也是同样的套路，qh 调用 dma\_pool\_alloc 来申请，然后用 me mset 给它清零。dma\_handle 同样传递给了 dma\_pool\_alloc，并且之后也赋值给了 qh->dma\_handle。qh 同样有一个成员 dma\_addr\_t dma\_handle。qh 中也有两个成员是队列：struct list\_head node 和 struct list\_head queue，同样也是在这里被初始化。此外，还有四个成员被赋了值，即 element、link、state 和 type。关于这四个赋值，我们暂时不提，用到了再说。不过我们应该回到 uhci\_start 的上下文去看一下 uhci\_alloc\_qh 被调用的具体情况，622 行这里有一个循环，UHCI\_NUM\_SKELQH 是一个宏，这个宏的值为 11，所以这里就是申请了 11 个 QH，这正是我们前面介绍过的那个

11 个 Skeleton QH。与此同时我们注意到 `struct uhci_hcd` 中有一个成员 `struct uhci_qh *skelqh[UHCI_NUM_SKELQH]`，即有这么一个数组，数组 11 个元素，而这里就算是为这 11 个元素申请了内存空间。

接下来，要具体解释这里的代码。还得看来自 `drivers/usb/host/uhci-hcd.h` 的一些宏：

```
317 #define UHCI_NUM_SKELQH      11
318 #define SKEL_UNLINK          0
319 #define skel_unlink_qh       skelqh[SKEL_UNLINK]
320 #define SKEL_ISO              1
321 #define skel_iso_qh          skelqh[SKEL_ISO]
322 /* int128, int64, ..., int1 = 2, 3, ..., 9 */
323 #define SKEL_INDEX(exponent) (9 - exponent)
324 #define SKEL_ASYNC            9
325 #define skel_async_qh        skelqh[SKEL_ASYNC]
326 #define SKEL_TERM            10
327 #define skel_term_qh         skelqh[SKEL_TERM]
328
329 /* The following entries refer to sublists of skel_async_qh */
330 #define SKEL_LS_CONTROL       20
331 #define SKEL_FS_CONTROL       21
332 #define SKEL_FSBR              SKEL_FS_CONTROL
333 #define SKEL_BULK              22
```

好家伙，光注释就看得我一愣一愣的，可惜还是没看懂。但基本上我们能感觉出，当前我们的目标是为了建立 QH 数据结构，并把相关的队列给连接起来。

633 行，`SKEL_ISO` 是 1，`SKEL_ASYNC` 是 9，所以这里就是循环 7 次。实际上，在这 11 个元素的数组中，2 到 9 就是对应于中断传输的那 8 个 Skeleton QH，所以这里就是为这 8 个 QH 的 `link` 元素赋值。对于这 8 个 QH，周期为 128 ms 的那个 QH 被称为 `int128`，周期为 64 ms 的被称为 `int64`，于是就有了 `int128`，`int64`，`...`，`int1`，分别对应这个数组的 2，3，`...`，9 号元素。今后我们对这几个 QH 的称呼也是如此，`skel int128 QH`，`skel int64 QH`，`...`，`skel int2 QH`，`skel int1 QH`。

而这里我们还看到另一个家伙，`skel_async_qh`。它表示 `async queue`，指的是 `low-speed control`，`full-speed control`，`bulk` 这三种异步队列。与之对应的就是刚才的 `SKEL_ASYNC` 宏，`SKEL_ASYNC` 等于 9，而我们同时知道 `skel int1 QH` 实际上也是 `skelqh[]` 数组的 9 号元素，所以实际上 `skel_async_qh` 和 `skel int1 QH` 是共用了同一个 QH，这是因为 `skel int1 QH` 表示中断传输的周期为 1 ms，而控制传输和批量传输也是每一个 ms 或者说每一个 Frame 都会被调度的，当然前提是带宽足够。所以这里的做法就是把 `skel int128 QH` 到 `skel int2 QH` 的 `link` 指针全都赋为 `LINK_TO_QH(uhci->skel_async_qh)`。

`LINK_TO_QH` 是一个宏，定义于 `drivers/usb/host/uhci-hcd.h`：

```
174 #define LINK_TO_QH(qh) \
    (UHCI_PTR_QH | cpu_to_le32((qh)->dma_handle))
```

UHCI\_PTR\_QH 等一系列宏也来自同一文件：

```
76 #define UHCI_PTR_BITS          __constant_cpu_to_le32(0x000F)
77 #define UHCI_PTR_TERM          __constant_cpu_to_le32(0x0001)
78 #define UHCI_PTR_QH            __constant_cpu_to_le32(0x0002)
79 #define UHCI_PTR_DEPTH         __constant_cpu_to_le32(0x0004)
80 #define UHCI_PTR_BREADTH       __constant_cpu_to_le32(0x0000)
```

这样我们就要看 `struct uhci_qh` 这个结构体中的成员 `__le32 link` 了。这是一个指针，这个指针指向下一个 QH，换言之，它包含着下一个 QH 或者下一个 TD 的地址。不过它一共 32 个 bits，其中只有 bit31 到 bit4 这些位是用来记录地址的，bit3 和 bit2 是保留位，bit1 则用来表示该指针指向的是一个 QH 还是一个 TD。bit1 如果为 1，表示本指针指向的是一个 QH，如果为 0，表示本指针指向的是一个 TD。（刚才这个宏 `UHCI_PTR_QH` 正是起到这个作用的，实际上 QH 总是 16 字节对齐的，即它的低四位总是为 0，所以我们总是把低四位拿出来利用，比如这里的 `LINK_TO_QH` 就是把这个 `struct uhci_qh` 的 bit1 给设置成 1，以表明它指向的是一个 QH。）而 bit0 表示本 QH 是否是最后一个 QH，如果 bit0 位 1，则说明本 QH 是最后一个 QH 了，所以这个指针实际上是无效的，而 bit0 为 0 才表示本指针有效，因为至少 QH 后面还有 QH 或者还有 TD。我们看到 `skel_async_qh` 的 link 指针被赋予了 `UHCI_PTR_TERM`。

另外这里还为 `skel_term_qh` 的 link 给赋了值，我们看到它就指向自己。`skel_term_qh` 是 `skelqa[]` 数组的第 10 个元素，其作用暂时还不明了。但以后自然会知道的。

`struct uhci_td` 里面同样也有个指针，`__le32 link`，它同样指向另一个 TD 或者 QH，而 bit1 和 bit0 的作用和 `struct uhci_qh` 中的 link 是一模一样的，bit1 为 1 表示指向 QH，为 0 表示指向 TD。bit0 为 1 表示指针无效，即本 TD 是最后一个 TD 了，bit0 为 0 表示指针有效。

639 行 `uhci_fill_td`，来自 `drivers/usb/host/uhci-q.c`：

```
138 static inline void uhci_fill_td(struct uhci_td *td, u32 status,
139                               u32 token, u32 buffer)
140 {
141     td->status = cpu_to_le32(status);
142     td->token = cpu_to_le32(token);
143     td->buffer = cpu_to_le32(buffer);
144 }
```

实际上就是填充 `struct uhci_td` 中的三个成员：`__le32 status`，`__le32 token` 和 `__le32 buffer`。咱们来看传递给它的参数，`status` 和 `buffer` 都是 0，只有一个 `token` 不为 0，`uhci_explen` 来自 `drivers/usb/host/uhci-hcd.h`：

```
211 #define td_token(td)            le32_to_cpu((td)->token)
212 #define TD_TOKEN_DEVADDR_SHIFT  8
213 #define TD_TOKEN_TOGGLE_SHIFT  19
214 #define TD_TOKEN_TOGGLE         (1 << 19)
215 #define TD_TOKEN_EXPLEN_SHIFT   21
216 #define TD_TOKEN_EXPLEN_MASK    0x7FF /* expected length, encoded as n-1*/
217 #define TD_TOKEN_PID_MASK       0xFF
218
```

```

219 #define uhci_explen(len)      (((len) - 1) & TD_TOKEN_EXPLEN_MASK) << \
220                               TD_TOKEN_EXPLEN_SHIFT)
221
222 #define uhci_expected_length(token) (((token) >> TD_TOKEN_EXPLEN_SHIFT) + \
223                                     1) & TD_TOKEN_EXPLEN_MASK)
224 #define uhci_toggle(token)    (((token) >> TD_TOKEN_TOGGLE_SHIFT) & 1)
225 #define uhci_endpoint(token)  (((token) >> 15) & 0xf)
226 #define uhci_devaddr(token)  (((token) >> TD_TOKEN_DEVADDR_SHIFT) & 0x7f)
227 #define uhci_devep(token)    (((token) >> TD_TOKEN_DEVADDR_SHIFT) & 0x7ff)
228 #define uhci_packetid(token)  ((token) & TD_TOKEN_PID_MASK)
229 #define uhci_packetout(token) (uhci_packetid(token) != USB_PID_IN)
230 #define uhci_packetin(token)  (uhci_packetid(token) == USB_PID_IN)

```

真是一波未平一波又起。麻烦的东西一个又一个地跳出来。让我一次次地感到心力交瘁。关于 token, UHCI spec 为 TD 定义了 4 个双字, 即四个 DWord, 其中第三个 DWord 叫做 TD TOKEN。一个 DWord 一共 32 个 bits。在这 32 个 bits 中, bit31 到 bit21 表示 Maximum Length, 即这次传输的最大允许字节。bit20 是保留位, bit19 表示 Data Toggle, bit18 到 bit15 表示端点的地址, 即我们曾经说的端点号; bit14 到 bit8 表示设备地址, bit7 到 bit0 表示 PID, 即 Packet ID。以上这些宏就是为了提取出一个 Token 的各个部分, 比如 uhci\_toggle, 就是 token 右移 19 位, 然后和 1 相与, 结果当然就是 token 的 bit19, 正是刚才说的 Data Toggle。而 uhci\_expected\_length 则是获取 bit31 到 bit21, 即 Length 这一段(加上 1 是因为 Spec 规定, 这一段为 0 表示 1 个 Byte, 为 1 表示 2 个 Bytes, 为 2 表示 3 个 Bytes……)

于是我们很快就能明白这个 uhci\_fill\_td 具体做了什么。(0x7f << TD\_TOKEN\_DEVADDR\_SHIFT)表示把 7f 左移 8 位, USB\_PID\_IN 等于 0x69, UHCI spec 规定这就表示 PID IN。然后 uhci\_explen(len)的作用和 uhci\_expected\_length 的作用恰恰相反, 它把一个 length 转换成 bit31 到 bit21, 这样三块“或”一下, 就构造了一个新的 token。

至于这个 token 构造好了之后填充给这 TD 究竟有什么用, 我们看不出来, 实际上注释说了, 这是为了修复一个 Bug, 若干年前, Intel PIIX 控制器有一个 Bug, 当时为了绕开这个 Bug, 引入了这么一段。

关于这个 Bug 的详情, 我们在后面会讲, 它和一个叫做 FSBP 的东西有关。只不过我们现在看到的是 term\_td 的 link 指针被设置为了 UHCI\_PTR\_TERM, 和 skel\_term\_qh 的 link 赋值一样, 又是那个休止符。其实这里的道理很简单, 就相当于我们每次申请一个链表时总是把最后一个指针设置成 NULL 一样。只不过这里不是叫作 NULL, 是叫作 UHCI\_PTR\_TERM, 但其作用都是一样, 就相当于五线谱中的休止符。注意 uhci->term\_td 正是我们一开始调用 uhci\_alloc\_td 时申请并且做的初始化。

642 行, struct uhci\_qh 中另一个成员为 \_\_le32 element。它指向一个队列中的第一个元素。LINK\_TO\_TD 来自 drivesr/usb/host/uhci-hcd.h:

```

269 #define LINK_TO_TD(td)      (cpu_to_le32((td)->dma_handle))

```

理解了 LINK\_TO\_QH 自然就能理解 LINK\_TO\_TD。这里咱们令 skel\_async\_qh 和

skel\_term\_qh 的 element 等于这个 uhci->term\_td。

649 行，这个循环可够夸张的，UHCI\_NUMFRAMES 的值为 1024，所以这个循环就是“惊世骇俗”的 1024 次。于是写代码的人说：“我把循环次数看成是一个程序高效的标志，是一件值得欣喜的事情。如果一个程序没有循环，那它的效率也可能惨不忍睹……”

## 12. 一个函数引发的故事（四）

uhci\_frame\_skel\_link 函数来自 drivers/usb/host/uhci-hcd.c:

```

97 static __le32 uhci_frame_skel_link(struct uhci_hcd *uhci, int frame)
98 {
99     int skelnum;
100
101     /*
102      * The interrupt queues will be interleaved as evenly as possible.
103      * There's not much to be done about period-1 interrupts; they have
104      * to occur in every frame. But we can schedule period-2 interrupts
105      * in odd-numbered frames, period-4 interrupts in frames congruent
106      * to 2 (mod 4), and so on. This way each frame only has two
107      * interrupt QHs, which will help spread out bandwidth utilization.
108      *
109      * ffs (Find First bit Set) does exactly what we need:
110      * 1,3,5,... => ffs = 0 => use period-2 QH = skelqh[8],
111      * 2,6,10,... => ffs = 1 => use period-4 QH = skelqh[7], etc.
112      * ffs >= 7 => not on any high-period queue, so use
113      *     period-1 QH = skelqh[9].
114      * Add in UHCI_NUMFRAMES to insure at least one bit is set.
115      */
116     skelnum = 8 - (int) __ffs(frame | UHCI_NUMFRAMES);
117     if (skelnum <= 1)
118         skelnum = 9;
119     return LINK_TO_QH(uhci->skelqh[skelnum]);
120 }

```

俗话说，彪悍的人生不需要解释，“彪悍”的代码不需要注释。但是像这个函数这样，仅仅几行代码，却用了一堆的注释，不可谓不“彪悍”也。

首先\_\_ffs 是一个位操作函数，其意思已经在注释里说得很清楚了，给它一个输入参数，它为你找到这个输入参数的第一个被 set 的位，被 set 就是说为 1。这个函数涉及汇编代码，对我这个汇编语言不会编、微机原理闹危机的人来说，显然是不愿意仔细去看这个函数具体是怎么实现的，只是知道，每个体系结构都实现了自己的这个函数\_\_ffs。比如，i386 的就在 include/asm-i386/bitops.h 中，而 x86 64 位的就在 include/asm-x86\_64/bitops.h 中。

在 Intel 以结果为导向的理念指导下，我们来看一下这个函数的返回值，很显然，正如注释里说的那样，如果输入参数是 1, 3, 5, 7, 9 等奇数，那么返回值必然是 0，因为 bit0 肯定是



1。如果参数是 2, 6, 10, 14, 18, 22, 26 这一系列的数, 那么返回值就是 1, 因为 bit0 一定是 0, 而 bit1 一定是 1。如果参数是 4, 12, 20, 28, 36 这一系列的数, 那么返回值就是 2, 因为 bit0 和 bit1 一定是 0, 而 bit2 一定是 1。

不难看出, 其实第一组数就是除以 2 余数为 1 的数列, 第二组数就是除以 4 余数为 2 的数列, 第三组就是除以 8 余数为 4 的数列, 用当年奥赛的术语取模符号 (mod) 来说, 就是第一组是  $1 \bmod 2$ , 第二组是  $2 \bmod 4$ , 第三组是  $4 \bmod 8$ , 如此下去, 返回值为 0 的一共有 512 个, 返回值为 1 的一共有 256 个, 返回值为 2 的一共有 128 个, 返回值为 3 的一共有 64 个, 返回值为 4 的一共有 32 个, 返回值为 5 的一共有 16 个, 返回值为 6 的一共有 8 个, 返回值为 7 的一共有 4 个, 返回值为 8 的一共有 2 个, 返回值为 9 的一共有 1 个(即 512)。N 年前我们就知道,  $1+2+4+\dots+512=1023$ 。

结合咱们这里代码的 116 行, frame 为 0 的话, \_\_ffs 的返回值为 10, 所以 skelnum 就应该为 -2, frame 为 1 到 1023 这个过程中, skelnum 为 8 的次数为 512, 为 7 的次数为 256, 为 6 的次数为 128, 为 5 的次数为 64, 为 4 的次数为 32, 为 3 的次数为 16, 为 2 的次数为 8, 为 1 的次数为 4, 为 0 的次数为 2, 为 -1 的次数为 1。而 117 行这个 if 语句就使得 skelnum 小于等于 1 的那几次都把 skelnum 设置为 9, 这总共有 8 次。(为 1 的 4 次, 为 0 的 2 次, 为 -1 的 1 次, 为 -2 的一次。)

因此我们就知道 skelnum 的取值范围是 2 到 9, 而这就意味着这里 uhci\_frame\_skel\_link 函数的返回值实际上就是 uhci->skelqh[] 这个数组中的 7 个元素。前面已经知道这个数组一共有 11 个元素, 除了 skelqh[2] 到 skelqh[9] 这 8 个元素以外, skelqh[1] 是为等时传输准备的, skel[10] 是休止符 (即 skel\_term\_qh), skel[0] 表示没有连接的状态。

要深刻理解这个数组需要时间的沉淀, 但是很明显, uhci\_frame\_skel\_link 的效果就是为 skelqh[2] 和 skelqh[9] 找到了归宿。在 1024 个 frame 中, 有 8 个 frame 指向了 skelqh[2], 即 skel int128 QH, 1024 除以 8 等于 128, 岂不正是每隔 128 ms 这个 qh 会被访问到么? 同理, 16 个 frame 指向了 skelqh[3], 即 skel int64 QH, 1024 除以 16 等于 64, 也正意味着每隔 64 ms 会被访问到。一直到 skelqh[8], 即 skel int2 QH, 有 512 个 frame 指向了它, 所以这就代表每隔 2 ms 会被访问的那个队列。剩下的 skelqh[9], 即 skel int1 QH, 总共也有 8 次, 不过你别误会, skel int1 QH 代表的是 1 ms 周期, 显然应该是 1024 个 frame 都指向它。可是你别忘了, 刚才我们不是把 skel int2 QH 到 skel int8 QH 的 link 指针都指向了 skel int1 QH 了么?

还没明白? 我们说了要用图解法来理解这个问题, 所以不妨先画出此时此刻这整个框架。

```

framelist[]
[ 0 ]----> QH -----\
[ 1 ]----> QH -----> QH ----> UHCI_PTR_TERM
...      ----> QH -----\
[1023]----> QH -----/
              ^^          ^^          ^^
            7 QHs for    1 QH for    f    End Chain

```

INT (2-128 ms) 1 ms-INT(plus CTRL Chain,BULK Chain)

skel 实际上就是 Skeleton，框架或者骨骼的意思。skelqh 数组扮演着一个框架的作用。实际上一个 QH 对应着一个端点，即从主机控制器的角度来说，它为每一个端点建立一个队列，这就要求每个队列有一个队列头，而许多个端点的队列如何组织呢？正如上面显示的框架一样，有了一个端点，就为它建立相应的队列，根据需要来建立，然后把它插入到框架中的对应位置。

## 13. 一个函数引发的故事（五）

接着走，661 行，configure\_hc，来自 drivers/usb/host/uhci-hcd.c，

```
178 static void configure_hc(struct uhci_hcd *uhci)
179 {
180     /* Set the frame length to the default: 1 ms exactly */
181     outb(USBSOF_DEFAULT, uhci->io_addr + USBSOF);
182
183     /* Store the frame list base address */
184     outl(uhci->frame_dma_handle, uhci->io_addr + USBFLBASEADD);
185
186     /* Set the current frame number */
187     outw(uhci->frame_number & UHCI_MAX_SOF_NUMBER,
188         uhci->io_addr + USBFRNUM);
189
190     /* Mark controller as not halted before we enable interrupts */
191     uhci_to_hcd(uhci)->state = HC_STATE_SUSPENDED;
192     mb();
193
194     /* Enable PIRQ */
195     pci_write_config_word(to_pci_dev(uhci_dev(uhci)), USBLEGSUP,
196         USBLEGSUP_DEFAULT);
197 }
```

USBSOF\_DEFAULT 和 USBSOF 定义于 drivers/usb/host/uhci-hcd.h:

```
43 #define USBFRNUM        6
44 #define USBFLBASEADD    8
45 #define USBSOF          12
46 #define USBSOF_DEFAULT  64    /* Frame length is exactly 1 ms */
```

UHCI spec 中定义了一个 START OF FRAME(SOF) MODIFY REGISTER，这里称作 SOF 寄存器，其地址位于 Base+(0Ch)处，0Ch 即这里的 12。这个寄存器值的修改意味着 Frame 周期的改变，通常我们没有必要修改这个寄存器，直接设置为默认值 64 即可，按照 UHCI spec 中 2.1.6 的陈述，这意味着对于常见的 12MHz 的时钟输入的情况，Frame 周期将为 1 ms。(The default value is decimal 64 which gives a SOF cycle time of 12000. For a 12 MHz SOF counter clock input, this produces a 1 ms Frame period.)

紧接着 USBFLBASEADD 用来表示另一个寄存器，UHCI spec 中称之为 FLBASEADD，即

Frame List 基地址寄存器，它一共有 32 个 bits，位于 Base+(08-0Bh)，这个寄存器应该包含 Frame List 在系统内存中的起始地址。其中，bit31 到 bit12 对应于内存地址信号[31: 12]，而 bit11 到 bit0 则是保留位，必须全为 0。frame\_dma\_handle 正是前面调用 dma\_alloc\_coherent 为 uhci->frame 申请内存时映射的 DMA 地址，显然这个地址需要写到这个寄存器里，这样主机控制器才会知道去怎样访问这个 Frame List。

接下来，UHCI\_MAX\_SOF\_NUMBER 是定义于 drivers/usb/host/uhci-hcd.h 的宏，值为 2047，用二进制来表示就是 11 个 1，即 111 1111 1111，而从 USBFRNUM 这里我们看到了是 6，它对应于 UHCI spec 中的寄存器 FRNUM (Frame Number Register)，地址为 Base+(06-07h)，这个寄存器一共 16 个 bits，这其中 bit10 到 bit0 包含了当前的 Frame 号，所以把 uhci->frame\_number 与 UHCI\_MAX\_SOF\_NUMBER 相与就得到它的 bit10 到 bit0 这 11 个 bits，即得到 Frame 号然后写入到 FRNUM 寄存器中去。unsigned int frame\_number 是 struct uhci\_hcd 的一个成员。

然后，设置 uhci\_to\_hcd(uhci)->state 为 HC\_STATE\_SUSPENDED，注意我们当初在 finish\_reset 中也有设置过这个状态，只不过当时是设置成了 HC\_STATE\_HALT。凡事都是有因有果的，我们做了这些设置，到时候自然会用到的。

195 行，pci\_write\_config\_word，写寄存器，USBLEGSUP，不过这次写的是 USBLEGSUP\_DEFAULT，这个宏的值为 0x2000，这是 UHCI spec 中规定的默认值。

这样我们就算是配置好了 HC，到这里就算万事俱备，只欠东风了。662 行就设置 uhci->is\_initialized 为 1，这意图再明显不过了。

回到 uhci\_start 中，还剩下最后一个函数，start\_rh()，rh 表示 Root Hub。这个函数来自 drivers/usb/host/uhci-hcd.c:

```
324 static void start_rh(struct uhci_hcd *uhci)
325 {
326     uhci_to_hcd(uhci)->state = HC_STATE_RUNNING;
327     uhci->is_stopped = 0;
328
329     /* Mark it configured and running with a 64-Byte max packet.
330      * All interrupts are enabled, even though RESUME won't do anything.
331      */
332     outw(USBCMD_RS | USBCMD_CF | USBCMD_MAXP, uhci->io_addr + USBCMD);
333     outw(USBINTR_TIMEOUT | USBINTR_RESUME | USBINTR_IOC | USBINTR_SP,
334          uhci->io_addr + USBINTR);
335     mb();
336     uhci->rh_state = UHCI_RH_RUNNING;
337     uhci_to_hcd(uhci)->poll_rh = 1;
338 }
```

又一次设置了 uhci\_to\_hcd(uhci)->state，只不过这次设置的是 HC\_STATE\_RUNNING。之后设置 is\_stopped 为 0。

然后是写寄存器 USBCMD，这次写的是什么呢？先看 drivers/usb/host/uhci-hcd.h 中关于这

个命令寄存器定义的宏：

```
16 #define USBCMD          0
17 #define  USBCMD_RS      0x0001 /* Run/Stop */
18 #define  USBCMD_HCRESET 0x0002 /* Host reset */
19 #define  USBCMD_GRESET  0x0004 /* Global reset */
20 #define  USBCMD_EGSM    0x0008 /* Global Suspend Mode */
21 #define  USBCMD_FGR     0x0010 /* Force Global Resume */
22 #define  USBCMD_SWDBG    0x0020 /* SW Debug mode */
23 #define  USBCMD_CF       0x0040 /* Config Flag (sw only) */
24 #define  USBCMD_MAXP     0x0080 /* Max Packet (0 = 32, 1 = 64) */
```

结合 spec 和这里的注释来看，USBCMD\_RS 表示 RUN/STOP，1 表示 RUN，0 表示 STOP。USBCMD\_CF 表示 Configure Flag，在配置阶段结束时，应该把这个 flag 设置好。USBCMD\_MAXP 则表示 FSR 最大包的 size，这位为 1 表示 64Bytes，为 0 表示 32Bytes。关于 FSR 我们以后会知道。

然后写另一个寄存器，USBINTR，表示中断使能寄存器。这个寄存器我们前面曾经提过。当时我们贴出了关于它的图片，知道它的 bit3，bit2，bit1，bit0 分别表示四个开关，为 1 就是使能，为 0 就是使不能，drivers/usb/host/uhci-hcd.h 中也定义了这些相关的宏：

```
37 #define USBINTR          4
38 #define  USBINTR_TIMEOUT 0x0001 /* Timeout/CRC error enable */
39 #define  USBINTR_RESUME  0x0002 /* Resume interrupt enable */
40 #define  USBINTR_IOC     0x0004 /* Interrupt On Complete enable */
41 #define  USBINTR_SP      0x0008 /* Short packet interrupt enable */
```

显而易见的是，咱们这里就是把这四个开关全部打开。这四种中断的意思都在注释里说得很清楚了：第一种是超时，第二种是从 Suspended 进入 Resume，第三种是完成了一个交易，第四种是接收到的包小于预期的长度。在这四种情况下，USB 主机控制器可以向系统主机或者说向系统的 CPU 发送中断请求。

最后设置 uhci->rh\_state 为 UHCI\_RH\_RUNNINNG，并设置 uhci\_to\_hcd(uhci)->poll\_rh 为 1。到这里 uhci\_start 就可以返回了，没什么意外的话就返回 0。于是咱们还是回到 usb\_add\_hcd 中去。

## 14. 寂寞在唱歌

接下来就该是 usb\_hcd\_poll\_rh\_status 函数了，这个函数在咱们整个故事将出现多次，甚至可以说在任何一个 HCD 的故事中都将出现多次。为了继续走下去，我们必须做一个伟大的假设，假设现在 Root Hub 上还没有连接任何设备，也就是说此时此刻，USB 设备树上只有 Root Hub 形单影只。我们以此为上下文开始往下看。

usb\_hcd\_poll\_rh\_status 来自 drivers/usb/core/hcd.c:

```

541 void usb_hcd_poll_rh_status(struct usb_hcd *hcd)
542 {
543     struct urb      *urb;
544     int              length;
545     unsigned long    flags;
546     char             buffer[4];      /* Any root hubs with > 31 ports? */
547
548     if (unlikely(!hcd->rh_registered))
549         return;
550     if (!hcd->uses_new_polling && !hcd->status_urb)
551         return;
552
553     length = hcd->driver->hub_status_data(hcd, buffer);
554     if (length > 0) {
555
556         /* try to complete the status urb */
557         local_irq_save (flags);
558         spin_lock(&hcd_root_hub_lock);
559         urb = hcd->status_urb;
560         if (urb) {
561             spin_lock(&urb->lock);
562             if (urb->status == -EINPROGRESS) {
563                 hcd->poll_pending = 0;
564                 hcd->status_urb = NULL;
565                 urb->status = 0;
566                 urb->hcpriv = NULL;
567                 urb->actual_length = length;
568                 memcpy(urb->transfer_buffer, buffer, length);
569             } else /* urb has been unlinked */
570                 length = 0;
571             spin_unlock(&urb->lock);
572         } else
573             length = 0;
574         spin_unlock(&hcd_root_hub_lock);
575
576         /* local irqs are always blocked in completions */
577         if (length > 0)
578             usb_hcd_giveback_urb (hcd, urb);
579         else
580             hcd->poll_pending = 1;
581         local_irq_restore (flags);
582     }
583
584     /* The USB 2.0 spec says 256 ms. This is close enough and won't
585      * exceed that limit if HZ is 100. */
586     if (hcd->uses_new_polling ? hcd->poll_rh :
587         (length == 0 && hcd->status_urb != NULL))
588         mod_timer (&hcd->rh_timer, jiffies + msecs_to_jiffies(250));
589 }

```

这个函数天生是为了中断传输而活的。

前面两个 if 肯定是不满足的。rh\_registered 刚刚才被设置为 1， uses\_new\_polling 也在 uhci\_start()中设置为了 1。所以，继续昂首挺胸地往前走。

553 行, `driver->hub_status_data` 是每个驱动自己定义的, 对于 UHCI 来说, 定义了 `uhci_hub_status_data` 这么一个函数, 它来自 `drivers/usb/host/uhci-hub.c` 中:

```

184 static int uhci_hub_status_data(struct usb_hcd *hcd, char *buf)
185 {
186     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
187     unsigned long flags;
188     int status = 0;
189
190     spin_lock_irqsave(&uhci->lock, flags);
191
192     uhci_scan_schedule(uhci);
193     if (!test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags) || uhci->dead)
194         goto done;
195     uhci_check_ports(uhci);
196
197     status = get_hub_status_data(uhci, buf);
198
199     switch (uhci->rh_state) {
200     case UHCI_RH_SUSPENDING:
201     case UHCI_RH_SUSPENDED:
202         /* if port change, ask to be resumed */
203         if (status)
204             usb_hcd_resume_root_hub(hcd);
205         break;
206
207     case UHCI_RH_AUTO_STOPPED:
208         /* if port change, auto start */
209         if (status)
210             wakeup_rh(uhci);
211         break;
212
213     case UHCI_RH_RUNNING:
214         /* are any devices attached? */
215         if (!any_ports_active(uhci)) {
216             uhci->rh_state = UHCI_RH_RUNNING_NODEVS;
217             uhci->auto_stop_time = jiffies + HZ;
218         }
219         break;
220
221     case UHCI_RH_RUNNING_NODEVS:
222         /* auto-stop if nothing connected for 1 second */
223         if (any_ports_active(uhci))
224             uhci->rh_state = UHCI_RH_RUNNING;
225         else if (time_after_eq(jiffies, uhci->auto_stop_time))
226             suspend_rh(uhci, UHCI_RH_AUTO_STOPPED);
227         break;
228
229     default:
230         break;
231     }
232
233 done:
234     spin_unlock_irqrestore(&uhci->lock, flags);
235     return status;
236 }

```

坦白说，这个函数一下子就把咱们这个�事的技术含量给拉高了上去。尤其是那个 `uhci_scan_schedule`，一下子就让故事变得复杂了起来，原本清晰的故事，渐渐变得模糊。

`uhci_scan_schedule` 来自 `drivers/usb/host/uhci-q.c`:

```

1708 static void uhci_scan_schedule(struct uhci_hcd *uhci)
1709 {
1710     int i;
1711     struct uhci_qh *qh;
1712
1713     /* Don't allow re-entrant calls */
1714     if (uhci->scan_in_progress) {
1715         uhci->need_rescan = 1;
1716         return;
1717     }
1718     uhci->scan_in_progress = 1;
1719 rescan:
1720     uhci->need_rescan = 0;
1721     uhci->fsbr_is_wanted = 0;
1722
1723     uhci_clear_next_interrupt(uhci);
1724     uhci_get_current_frame_number(uhci);
1725     uhci->cur_iso_frame = uhci->frame_number;
1726
1727     /* Go through all the QH queues and process the URBs in each one */
1728     for (i = 0; i < UHCI_NUM_SKELQH - 1; ++i) {
1729         uhci->next_qh = list_entry(uhci->skelqh[i]->node.next,
1730                                   struct uhci_qh, node);
1731         while ((qh = uhci->next_qh) != uhci->skelqh[i]) {
1732             uhci->next_qh = list_entry(qh->node.next,
1733                                       struct uhci_qh, node);
1734
1735             if (uhci_advance_check(uhci, qh)) {
1736                 uhci_scan_qh(uhci, qh);
1737                 if (qh->state == QH_STATE_ACTIVE) {
1738                     uhci_urbp_wants_fsbr(uhci,
1739                                           list_entry(qh->queue.next, struct urb_priv, node));
1740                 }
1741             }
1742         }
1743     }
1744
1745     uhci->last_iso_frame = uhci->cur_iso_frame;
1746     if (uhci->need_rescan)
1747         goto rescan;
1748     uhci->scan_in_progress = 0;
1749
1750     if (uhci->fsbr_is_on && !uhci->fsbr_is_wanted &&
1751         !uhci->fsbr_expiring) {
1752         uhci->fsbr_expiring = 1;
1753         mod_timer(&uhci->fsbr_timer, jiffies + FSBR_OFF_DELAY);
1754     }
1755
1756     if (list_empty(&uhci->skel_unlink_qh->node))
1757         uhci_clear_next_interrupt(uhci);
1758     else
1759         uhci_set_next_interrupt(uhci);

```

```
1760 }
```

这个函数的复杂性让我哭都哭不出来。我做梦也没想到这个函数竟然会牵出一打函数来。

`scan_in_progress` 初值为 0，只有在这个函数中才会改变它的值。因为它本来就是被用来表征我们处于这个函数中，注释中也说了，使用这个变量的目的就是为了避免这个函数被嵌套调用。所以如果这里判断为 0，则 1718 行就立刻设置它为 1。反之如果已经不为 0 了，就设置 `need_rescan` 为 1，并且函数返回。

1720 行和 1721 行，设置 `need_rescan` 和 `fsbr_is_wanted` 都为 0。

1723 行，`uhci_clear_next_interrupt()` 来自 `drivers/usb/host/uhci-q.c`:

```
35 static inline void uhci_clear_next_interrupt(struct uhci_hcd *uhci)
36 {
37     uhci->term_td->status &= ~cpu_to_le32(TD_CTRL_IOC);
38 }
```

清中断。如果一个 TD 设置了 `TD_CTRL_IOC` 这个 flag，就表示该 TD 所在的 Frame 结束之后，USB 主机控制器将向 CPU 发送一次中断。在这里我们实际上不希望 `term_td` 结束所在的 Frame 发生任何中断，因为我们现在正在处理整个调度队列。

而接下来的另一个函数 `uhci_get_current_frame_number()` 则来自 `drivers/usb/host/uhci-hcd.c`:

```
441 static void uhci_get_current_frame_number(struct uhci_hcd *uhci)
442 {
443     if (!uhci->is_stopped) {
444         unsigned delta;
445
446         delta = (inw(uhci->io_addr + USBFRNUM) - uhci->frame_number) &
447                (UHCI_NUMFRAMES - 1);
448         uhci->frame_number += delta;
449     }
450 }
```

正如注释里说的那样，把寄存器中的值更新给 `uhci->frame_number`。

我们结合 1729 行和 1731 行来看，注意到这里判断的就是 `uhci->skelqh[]` 数组的每个成员的 `node` 队列。我们知道 `struct uhci_qh` 结构体有一个成员是 `node`，它代表的是一支队伍，在 `uhci_alloc_qh` 中我们事实上用 `INIT_LIST_HEAD` 宏初始化了这个队列，这个宏就是初始化一个空队列，即一个节点的 `next` 和 `prev` 指针都指向自己。所以很显然，`uhci->next_qh` 就等于 `uhci->skelqh[i]`。于是 1731 这个 `while` 循环就不会执行，因此，1728 开始的这个 `for` 循环也就没有什么作用。或者至少说，现在还不是它“做贡献”的时刻。待到山花浪漫时，`skelqh[]` 的 `node` 队列有内容了，我们自然还会再次回来看这个函数。

飘过了这个 `for` 循环 `uhci_scan_schedule` 函数。1748 行又把 `scan_in_progress` 设置为 0。

1750 行自然也不用说，`fsbr_is_on` 默认也是 0。所以暂时这里也不会执行。



至于 1756 行这段 if-else, `skel_unlink_qh` 实际上就是 `skelqh[0]`, 同样, 此时此刻它的 `node` 队列也是空的, 故 `list_empty` 是满足的, 因此 `uhci_clear_next_interrupt` 会再次被调用。

结束了 `uhci_scan_schedule`, 我们继续回到 `uhci_hub_status_data` 中来。

193 行, `HCD_FLAG_HW_ACCESSIBLE` 这个 flag 我们是设置过的, 就在 `usb_add_hcd` 中设置的。而 `uhci->dead` 在 `finish_reset` 中设置为 0。

接下来 `uhci_check_ports` 函数, 来自 `drivers/usb/host/uhci-hub.c`:

```

136 static void uhci_check_ports(struct uhci_hcd *uhci)
137 {
138     unsigned int port;
139     unsigned long port_addr;
140     int status;
141
142     for (port = 0; port < uhci->rh_numports; ++port) {
143         port_addr = uhci->io_addr + USBPORTSC1 + 2 * port;
144         status = inw(port_addr);
145         if (unlikely(status & USBPORTSC_PR)) {
146             if (time_after_eq(jiffies, uhci->ports_timeout)) {
147                 CLR_RH_PORTSTAT(USBPORTSC_PR);
148                 udelay(10);
149
150                 /* HP's server management chip requires
151                  * a longer delay. */
152                 if (to_pci_dev(uhci_dev(uhci))->vendor ==
153                     PCI_VENDOR_ID_HP)
154                     wait_for_HP(port_addr);
155
156                 /* If the port was enabled before, turning
157                  * reset on caused a port enable change.
158                  * Turning reset off causes a port connect
159                  * status change. Clear these changes. */
160                 CLR_RH_PORTSTAT(USBPORTSC_CSC | USBPORTSC_PEC);
161                 SET_RH_PORTSTAT(USBPORTSC_PE);
162             }
163         }
164         if (unlikely(status & USBPORTSC_RD)) {
165             if (!test_bit(port, &uhci->resuming_ports)) {
166
167                 /* Port received a wakeup request */
168                 set_bit(port, &uhci->resuming_ports);
169                 uhci->ports_timeout = jiffies +
170                                     msecs_to_jiffies(20);
171
172                 /* Make sure we see the port again
173                  * after the resuming period is over. */
174                 mod_timer(&uhci->hcd(uhci)->rh_timer,
175                         uhci->ports_timeout);
176             } else if (time_after_eq(jiffies,
177                                     uhci->ports_timeout)) {
178                 uhci_finish_suspend(uhci, port, port_addr);
179             }
180         }
181     }

```

182 }

这个函数倒是挺清晰的，遍历 Root Hub 的各个端口，读取它们对应的端口寄存器。和这个端口寄存器相关的宏又很多，来自 `drivers/usb/host/uhci-hcd.h`:

```

49 #define USBPORTSC1      16
50 #define USBPORTSC2      18
51 #define USBPORTSC_CCS    0x0001 /* Current Connect Status
52                                * ("device present") */
53 #define USBPORTSC_CSC    0x0002 /* Connect Status Change */
54 #define USBPORTSC_PE     0x0004 /* Port Enable */
55 #define USBPORTSC_PEC    0x0008 /* Port Enable Change */
56 #define USBPORTSC_DPLUS  0x0010 /* D+ high (line status) */
57 #define USBPORTSC_DMINUS 0x0020 /* D- high (line status) */
58 #define USBPORTSC_RD     0x0040 /* Resume Detect */
59 #define USBPORTSC_RES1   0x0080 /* reserved, always 1 */
60 #define USBPORTSC_LSDA   0x0100 /* Low Speed Device Attached */
61 #define USBPORTSC_PR     0x0200 /* Port Reset */
62 /* OC and OCC from Intel 430TX and later (not UHCI 1.1d spec) */
63 #define USBPORTSC_OC     0x0400 /* Over Current condition */
64 #define USBPORTSC_OCC    0x0800 /* Over Current Change R/WC */
65 #define USBPORTSC_SUSP   0x1000 /* Suspend */
66 #define USBPORTSC_RES2   0x2000 /* reserved, write zeroes */
67 #define USBPORTSC_RES3   0x4000 /* reserved, write zeroes */
68 #define USBPORTSC_RES4   0x8000 /* reserved, write zeroes */

```

首先要看的是状态位 `USBPORTSC_PR`，为 1 表示此时此刻该端口正处于 `reset` 的状态。

其次我们看状态位 `USBPORTSC_RD`，这位为 1 表示端口探测到了 `resume` 信号。

显然以上两种情况都不是我们需要考虑的，至少不是现在需要考虑的。

于是下一个函数，`get_hub_status_data`，来自 `drivers/usb/host/uhci-hub.c`:

```

55 static inline int get_hub_status_data(struct uhci_hcd *uhci, char *buf)
56 {
57     int port;
58     int mask = RWC_BITS;
59
60     /* Some boards (both VIA and Intel apparently) report bogus
61      * overcurrent indications, causing massive log spam unless
62      * we completely ignore them. This doesn't seem to be a problem
63      * with the chipset so much as with the way it is connected on
64      * the motherboard; if the overcurrent input is left to float
65      * then it may constantly register false positives. */
66     if (ignore_oc)
67         mask &= ~USBPORTSC_OCC;
68
69     *buf = 0;
70     for (port = 0; port < uhci->rh_numports; ++port) {
71         if ((inw(uhci->io_addr + USBPORTSC1 + port * 2) & mask) ||
72             test_bit(port, &uhci->port_c_suspend))
73             *buf |= (1 << (port + 1));
74     }
75     return !!*buf;
76 }

```

这里 RWC\_BITS 就是用来屏蔽端口寄存器中的 RWC 的 bits。这三个都是状态改变位。

这里的 ignore\_oc 又是一个模块参数，uhci-hcd 和 ehci-hcd 这两个模块都会使用这个参数。注释里说得很清楚为何要用这个，有些主板喜欢“谎报军情”，对于这种情况，可以使用一个 ignore\_oc 参数来忽略之。

接下来又是遍历端口，读取每个端口的寄存器，如果有戏，就把信息存在 buf 中，直到这时我们才注意到 usb\_hcd\_poll\_rh\_status 函数中定义了一个 char buffer[4]，这个 buffer 被一次次地传递下来。buf 一共 32 个 bits，这里凡是一个端口的寄存器里有东西（除了状态改变位以外），就在 buf 里把相应的位设置为 1。如果设置了 port\_c\_suspend 也需要这样，port\_c\_suspend 到时我们在电源管理部分会看到，现在当然没有被设置。

不过这个函数最酷的就是最后这句居然有两个感叹号。这保证返回值要么是 0，要么是非 0。

接下来判断 uhci->rh\_state 了，前面在 start\_rh 中设置了它为 UHCI\_RH\_RUNNING，所以这里就是执行 any\_ports\_active。

这个 any\_ports\_active 也来自 drivers/usb/host/uhci-hub.c:

```
42 static int any_ports_active(struct uhci_hcd *uhci)
43 {
44     int port;
45
46     for (port = 0; port < uhci->rh_numports; ++port) {
47         if ((inw(uhci->io_addr + USBPORTSC1 + port * 2) &
48             (USBPORTSC_CCS | RWC_BITS)) ||
49             test_bit(port, &uhci->port_c_suspend))
50             return 1;
51     }
52     return 0;
53 }
```

这时再次读端口寄存器。其中 CCS 表示端口连接有变化。我们假设现在没有变化。那么这里什么也不干，直接返回 0。这样 uhci\_hub\_status\_data 最终返回 status。这个 status 正是 get\_hub\_status\_data 的返回值，即那个要么是 0，要么是非 0 的。

如果为 0，那么很好办，直接“凌波微步”来到 586 行，判断 hcd->uses\_new\_polling，咱们在 uhci\_start 中设置为了 1。所以这里继续判断 hcd->poll\_rh，在 start\_rh 中也把它设置为了 1。所以，这里 mod\_timer 会被执行。这个函数意味着时间到了某件事情就会发生，咱们曾经在 usb\_create\_hcd 中初始化过 hcd->rh\_timer，并且为它绑定了函数 rh\_timer\_func，所以不妨来看一下 rh\_timer\_func，来自 drivers/usb/core/hcd.c:

```
593 static void rh_timer_func (unsigned long _hcd)
594 {
595     usb_hcd_poll_rh_status((struct usb_hcd *) _hcd);
596 }
```

原来就是调用 `usb_hcd_poll_rh_status`。所以 `usb_hcd_poll_rh_status` 函数是一个被多次调用的函数，只不过多次之间是有个延时的，而咱们这里调用 `mod_timer` 设置的是 250 ms。而每次所做的就是去询问 Root Hub 的状态。实际上这就是 poll 的含义——轮询。

那么咱们这个故事基本上就结束了。我们能看到的是 `usb_hcd_poll_rh_status` 这个函数，每隔 250 ms 这样被执行一遍，重复一遍又一遍，可是它忙忙碌碌却什么也不做，但即便如此也比我要好，要知道我的人生就像复印机，每天都在不停地重复，可问题是有时候还卡纸。

## 15. Root Hub 的控制传输（一）

虽然最伟大的 `probe` 函数就这样结束了。但是，我们的道路还很长，困难还很多，最终的结局是未知数。

在剩下的篇幅中，我们将围绕 `usb_submit_urb()` 函数展开。子曾经曰过：不吃饭的女人这世上也许还有好几个，不吃醋的女人却连一个也没有。我也曾经曰过：不遵循 USB spec 的 USB 设备这世上也许还有好几个，不调用 `usb_submit_urb()` 的 USB 设备驱动却连一个也没有。想必一路走来的兄弟们早就想知道神秘的 `usb_submit_urb()` 函数究竟是怎么“混”的吧？

不管是控制传输，还是中断传输，或是批量传输，又或者等时传输，设备驱动都一定会调用 `usb_submit_urb` 函数，只有通过它才能提交 urb。所以接下来就分类来看这个函数，查看四种传输分别是如何处理的。

不过我们仍然先假设还没有设备插入 Root Hub 吧。因为 Root Hub 始终是一个特殊的角色，它的特殊地位决定了我们必须特殊对待。Hub 只涉及两种传输方式：控制传输和中断传输。我们先来看控制传输，确切地说是先看 Root Hub 的控制传输。

还记得刚才在 `register_root_hub` 中那个 `usb_get_device_descriptor` 么？在 Hub 驱动中讲过，它会调用 `usb_get_descriptor`，而后者会调用 `usb_control_msg`，而 `usb_control_msg` 则调用 `usb_internal_control_msg`，然后 `usb_start_wait_urb` 会被调用，但最终会被调用的是 `usb_submit_urb()`。于是我们就来看一下 `usb_submit_urb()` 究竟何德何能让大家如此景仰，我们来看这个设备描述符究竟是如何获得的。

这个函数显然分量比较重，它来自 `drivers/usb/core/urb.c`：

```
220 int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
221 {
222     int                pipe, temp, max;
223     struct usb_device  *dev;
224     int                is_out;
225
```

```

226     if (!urb || urb->hcpriv || !urb->complete)
227         return -EINVAL;
228     if (!(dev = urb->dev) ||
229         (dev->state < USB_STATE_DEFAULT) ||
230         (!dev->bus) || (dev->devnum <= 0))
231         return -ENODEV;
232     if (dev->bus->controller->power.power_state.event != PM_EVENT_ON
233         || dev->state == USB_STATE_SUSPENDED)
234         return -EHOSTUNREACH;
235
236     urb->status = -EINPROGRESS;
237     urb->actual_length = 0;
238
239     /* Lots of sanity checks, so HCDs can rely on clean data
240      * and don't need to duplicate tests
241      */
242     pipe = urb->pipe;
243     temp = usb_pipetype(pipe);
244     is_out = usb_pipeout(pipe);
245
246     if (!usb_pipecontrol(pipe) && dev->state < USB_STATE_CONFIGURED)
247         return -ENODEV;
248
249     /* FIXME there should be a sharable lock protecting us against
250      * config/altsetting changes and disconnects, kicking in here.
251      * (here == before maxpacket, and eventually endpoint type,
252      * checks get made.)
253      */
254
255     max = usb_maxpacket(dev, pipe, is_out);
256     if (max <= 0) {
257         dev_dbg(&dev->dev,
258             "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
259             usb_pipeendpoint(pipe), is_out ? "out" : "in",
260             __FUNCTION__, max);
261         return -EMSGSIZE;
262     }
263
264     /* periodic transfers limit size per frame/uframe,
265      * but drivers only control those sizes for ISO.
266      * while we're checking, initialize return status.
267      */
268     if (temp == PIPE_ISOCHRONOUS) {
269         int n, len;
270
271         /* "high bandwidth" mode, 1-3 packets/uframe? */
272         if (dev->speed == USB_SPEED_HIGH) {
273             int mult = 1 + ((max >> 11) & 0x03);
274             max &= 0x07ff;
275             max *= mult;
276         }
277
278         if (urb->number_of_packets <= 0)
279             return -EINVAL;
280         for (n = 0; n < urb->number_of_packets; n++) {
281             len = urb->iso_frame_desc[n].length;
282             if (len < 0 || len > max)
283                 return -EMSGSIZE;
284             urb->iso_frame_desc[n].status = -EXDEV;

```

```

285         urb->iso_frame_desc[n].actual_length = 0;
286     }
287 }
288
289 /* the I/O buffer must be mapped/unmapped, except when length=0 */
290 if (urb->transfer_buffer_length < 0)
291     return -EMSGSIZE;
292
293 #ifdef DEBUG
294 /* stuff that drivers shouldn't do, but which shouldn't
295  * cause problems in HCDs if they get it wrong.
296  */
297 {
298     unsigned int    orig_flags = urb->transfer_flags;
299     unsigned int    allowed;
300
301     /* enforce simple/standard policy */
302     allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
303               URB_NO_INTERRUPT);
304     switch (temp) {
305     case PIPE_BULK:
306         if (is_out)
307             allowed |= URB_ZERO_PACKET;
308         /* FALLTHROUGH */
309     case PIPE_CONTROL:
310         allowed |= URB_NO_FSBR; /* only affects UHCI */
311         /* FALLTHROUGH */
312     default:
313         /* all non-iso endpoints */
314         if (!is_out)
315             allowed |= URB_SHORT_NOT_OK;
316         break;
317     case PIPE_ISOCHRONOUS:
318         allowed |= URB_ISO_ASAP;
319         break;
320     }
321     urb->transfer_flags &= allowed;
322
323     /* fail if submitter gave bogus flags */
324     if (urb->transfer_flags != orig_flags) {
325         err("BOGUS urb flags, %x --> %x",
326             orig_flags, urb->transfer_flags);
327         return -EINVAL;
328     }
329 #endif
330 /*
331  * Force periodic transfer intervals to be legal values that are
332  * a power of two (so HCDs don't need to).
333  *
334  * FIXME want bus->{intr,iso}_sched_horizon values here. Each HC
335  * supports different values... this uses EHCI/UHCI defaults (and
336  * EHCI can use smaller non-default values).
337  */
338 switch (temp) {
339 case PIPE_ISOCHRONOUS:
340 case PIPE_INTERRUPT:
341     /* too small? */
342     if (urb->interval <= 0)
343         return -EINVAL;

```

```

344         /* too big? */
345         switch (dev->speed) {
346             case USB_SPEED_HIGH: /* units are microframes */
347                 // NOTE usb handles 2^15
348                 if (urb->interval > (1024 * 8))
349                     urb->interval = 1024 * 8;
350                 temp = 1024 * 8;
351                 break;
352             case USB_SPEED_FULL: /* units are frames/ msec */
353             case USB_SPEED_LOW:
354                 if (temp == PIPE_INTERRUPT) {
355                     if (urb->interval > 255)
356                         return -EINVAL;
357                     // NOTE ohci only handles up to 32
358                     temp = 128;
359                 } else {
360                     if (urb->interval > 1024)
361                         urb->interval = 1024;
362                     // NOTE usb and ohci handle up to 2^15
363                     temp = 1024;
364                 }
365                 break;
366             default:
367                 return -EINVAL;
368         }
369         /* power of two? */
370         while (temp > urb->interval)
371             temp >>= 1;
372         urb->interval = temp;
373     }
374
375     return usb_hcd_submit_urb(urb, mem_flags);
376 }

```

天哪，这个函数绝对够让你我看得“七窍流血”。这种变态已经不能用语言来形容了，鲁迅先生看了一定会说我已经出离愤怒了！南唐的李煜在看完这段代码之后感慨道：问君能有几多愁，恰似太監上青樓！

这个函数的核心变量就是那个 `temp`。很明显，它表示的就是传输管道的类型。我们说了现在考虑的是 Root Hub 的控制传输。那么很明显的事实是，`usb_hcd_submit_urb` 会被调用，而 268 行这个 `if` 语段和 338 行这个 `switch` 都没有什么意义。所以我们来看 `usb_hcd_submit_urb` 吧，来自 `drivers/usb/core/hcd.c`：

```

921 int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
922 {
923     int                status;
924     struct usb_hcd     *hcd = bus_to_hcd(urb->dev->bus);
925     struct usb_host_endpoint *ep;
926     unsigned long      flags;
927
928     if (!hcd)
929         return -ENODEV;
930
931     usbmon_urb_submit(&hcd->self, urb);
932

```

```

933      /*
934       * Atomically queue the urb, first to our records, then to the HCD.
935       * Access to urb->status is controlled by urb->lock ... changes on
936       * i/o completion (normal or fault) or unlinking.
937       */
938
939      // FIXME: verify that quiescing hc works right (RH cleans up)
940
941      spin_lock_irqsave (&hcd_data_lock, flags);
942      ep = (usb_pipein(urb->pipe) ? urb->dev->ep_in : urb->dev->ep_out)
943           [usb_pipeendpoint(urb->pipe)];
944      if (unlikely (!ep))
945          status = -ENOENT;
946      else if (unlikely (urb->reject))
947          status = -EPERM;
948      else switch (hcd->state) {
949          case HC_STATE_RUNNING:
950          case HC_STATE_RESUMING:
951      doit:
952          list_add_tail (&urb->urb_list, &ep->urb_list);
953          status = 0;
954          break;
955          case HC_STATE_SUSPENDED:
956              /* HC upstream links (register access, wakeup signaling) can work
957               * even when the downstream links (and DMA etc) are quiesced; let
958               * usbcore talk to the root hub.
959               */
960              if(hcd->self.controller->power.power_state.event==PM_EVENT_ON
961                  && urb->dev->parent == NULL)
962                  goto doit;
963              /* FALL THROUGH */
964          default:
965              status = -ESHUTDOWN;
966              break;
967      }
968      spin_unlock_irqrestore (&hcd_data_lock, flags);
969      if (status) {
970          INIT_LIST_HEAD (&urb->urb_list);
971          usbmon_urb_submit_error(&hcd->self, urb, status);
972          return status;
973      }
974
975      /* increment urb's reference count as part of giving it to the HCD
976       * (which now controls it). HCD guarantees that it either returns
977       * an error or calls giveback(), but not both.
978       */
979      urb = usb_get_urb (urb);
980      atomic_inc (&urb->use_count);
981
982      if (urb->dev == hcd->self.root_hub) {
983          /* NOTE: requirement on hub callers (usbfs and the hub
984           * driver, for now) that URBs' urb->transfer_buffer be
985           * valid and usb_buffer_{sync,unmap}() not be needed, since
986           * they could clobber root hub response data.
987           */
988          status = rh_urb_enqueue (hcd, urb);
989          goto done;
990      }
991

```



```

992     /* lower level hcd code should use *_dma exclusively,
993     * unless it uses pio or talks to another transport.
994     */
995     if (hcd->self.uses_dma) {
996         if (usb_pipecontrol (urb->pipe)
997             && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
998             urb->setup_dma = dma_map_single (
999                 hcd->self.controller,
1000                 urb->setup_packet,
1001                 sizeof (struct usb_ctrlrequest),
1002                 DMA_TO_DEVICE);
1003         if (urb->transfer_buffer_length != 0
1004             && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
1005             urb->transfer_dma = dma_map_single (
1006                 hcd->self.controller,
1007                 urb->transfer_buffer,
1008                 urb->transfer_buffer_length,
1009                 usb_pipein (urb->pipe)
1010                     ? DMA_FROM_DEVICE
1011                     : DMA_TO_DEVICE);
1012     }
1013
1014     status = hcd->driver->urb_enqueue (hcd, ep, urb, mem_flags);
1015 done:
1016     if (unlikely (status)) {
1017         urb_unlink (urb);
1018         atomic_dec (&urb->use_count);
1019         if (urb->reject)
1020             wake_up (&usb_kill_urb_queue);
1021         usbmon_urb_submit_error(&hcd->self, urb, status);
1022         usb_put_urb (urb);
1023     }
1024     return status;
1025 }

```

凡是名字中带着 `usbmon` 的函数都甭管，它是一个 USB 的监控工具，启用与否取决于一个编译选项：`CONFIG_USB_MON`，咱们假设不打开它，这样它的这些函数实际上就都是些空函数。就比如 931 行的 `usbmon_urb_submit`，以及下面的这个 `usbmon_urb_submit_error`。

942 行得到与这个 `urb` 相关的 `struct usb_host_endpoint` 结构体指针 `ep`。事实上 `struct urb` 和 `struct usb_host_endpoint` 这两个结构体中都有一个成员 `struct list_head urb_list`，每个端点都维护着一个队列，所有与它相关的 `urb` 都被放入到这个队列中，而 952 行所做的就是这件事。当然，之所以我们现在会执行 952 行，是因为我们的 `hcd->state` 在 `start_rh` 中被设置成了 `HC_STATE_RUNNING`。

接着，我们发现，对于 Root Hub，`rh_urb_enqueue` 会被执行；对于非 Root Hub，即一般的 Hub，`driver->urb_enqueue` 会被执行；对于 UHCI 来说，就是 `uhci_urb_enqueue` 会被执行。先来看 Root Hub。

`rh_urb_enqueue` 来自 `drivers/usb/core/hcd.c`：

```

629 static int rh_urb_enqueue (struct usb_hcd *hcd, struct urb *urb)

```

```

630 {
631     if (usb_pipeint (urb->pipe))
632         return rh_queue_status (hcd, urb);
633     if (usb_pipecontrol (urb->pipe))
634         return rh_call_control (hcd, urb);
635     return -EINVAL;
636 }

```

## 16. Root Hub 的控制传输（二）

对于控制传输，`rh_call_control` 会被调用。我也特别希望能有人给这个函数“吸吸脂”。我们的上下文是为了获取设备描述符，即当初那个 `usb_get_device_descriptor` 领着我们来到了这个函数，为了完成这件事情，实际上只需要很少的代码，但是 `rh_call_control` 这个函数涉及了所有与 Root Hub 相关的控制传输，以至于我们不得不顺便查看其他代码。当然了，既然是顺便，那么我们就不会详细地去讲解每一行。这个函数定义于 `drivers/usb/core/hcd.c`：

```

344 static int rh_call_control (struct usb_hcd *hcd, struct urb *urb)
345 {
346     struct usb_ctrlrequest *cmd;
347     u16      typeReq, wValue, wIndex, wLength;
348     u8       *ubuf = urb->transfer_buffer;
349     u8       tbuf [sizeof (struct usb_hub_descriptor)]
350         __attribute__((aligned(4)));
351     const u8 *bufp = tbuf;
352     int      len = 0;
353     int      patch_wakeup = 0;
354     unsigned long flags;
355     int      status = 0;
356     int      n;
357
358     cmd = (struct usb_ctrlrequest *) urb->setup_packet;
359     typeReq = (cmd->bRequestType << 8) | cmd->bRequest;
360     wValue  = le16_to_cpu (cmd->wValue);
361     wIndex  = le16_to_cpu (cmd->wIndex);
362     wLength = le16_to_cpu (cmd->wLength);
363
364     if (wLength > urb->transfer_buffer_length)
365         goto error;
366
367     urb->actual_length = 0;
368     switch (typeReq) {
369
370     /* DEVICE REQUESTS */
371
372     /* The root hub's remote wakeup enable bit is implemented using
373      * driver model wakeup flags. If this system supports wakeup
374      * through USB, userspace may change the default "allow wakeup"
375      * policy through sysfs or these calls.
376      *
377      * Most root hubs support wakeup from downstream devices, for

```

```

378 * runtime power management (disabling USB clocks and reducing
379 * VBUS power usage). However, not all of them do so; silicon,
380 * board, and BIOS bugs here are not uncommon, so these can't
381 * be treated quite like external hubs.
382 *
383 * Likewise, not all root hubs will pass wakeup events upstream,
384 * to wake up the whole system. So don't assume root hub and
385 * controller capabilities are identical.
386 */
387
388 case DeviceRequest | USB_REQ_GET_STATUS:
389     tbuf [0] = (device_may_wakeup(&hcd->self.root_hub->dev)
390                 << USB_DEVICE_REMOTE_WAKEUP)
391                 | (1 << USB_DEVICE_SELF_POWERED);
392     tbuf [1] = 0;
393     len = 2;
394     break;
395 case DeviceOutRequest | USB_REQ_CLEAR_FEATURE:
396     if (wValue == USB_DEVICE_REMOTE_WAKEUP)
397         device_set_wakeup_enable(&hcd->self.root_hub->dev, 0);
398     else
399         goto error;
400     break;
401 case DeviceOutRequest | USB_REQ_SET_FEATURE:
402     if (device_can_wakeup(&hcd->self.root_hub->dev)
403         && wValue == USB_DEVICE_REMOTE_WAKEUP)
404         device_set_wakeup_enable(&hcd->self.root_hub->dev, 1);
405     else
406         goto error;
407     break;
408 case DeviceRequest | USB_REQ_GET_CONFIGURATION:
409     tbuf [0] = 1;
410     len = 1;
411     /* FALLTHROUGH */
412 case DeviceOutRequest | USB_REQ_SET_CONFIGURATION:
413     break;
414 case DeviceRequest | USB_REQ_GET_DESCRIPTOR:
415     switch (wValue & 0xff00) {
416     case USB_DT_DEVICE << 8:
417         if (hcd->driver->flags & HCD_USB2)
418             bufp = usb2_rh_dev_descriptor;
419         else if (hcd->driver->flags & HCD_USB11)
420             bufp = usb11_rh_dev_descriptor;
421         else
422             goto error;
423         len = 18;
424         break;
425     case USB_DT_CONFIG << 8:
426         if (hcd->driver->flags & HCD_USB2) {
427             bufp = hs_rh_config_descriptor;
428             len = sizeof hs_rh_config_descriptor;
429         } else {
430             bufp = fs_rh_config_descriptor;
431             len = sizeof fs_rh_config_descriptor;
432         }
433         if (device_can_wakeup(&hcd->self.root_hub->dev))
434             patch_wakeup = 1;
435         break;
436     case USB_DT_STRING << 8:

```

```

437         n = rh_string (wValue & 0xff, hcd, ubuf, wLength);
438         if (n < 0)
439             goto error;
440         urb->actual_length = n;
441         break;
442     default:
443         goto error;
444     }
445     break;
446 case DeviceRequest | USB_REQ_GET_INTERFACE:
447     tbuf [0] = 0;
448     len = 1;
449     /* FALLTHROUGH */
450 case DeviceOutRequest | USB_REQ_SET_INTERFACE:
451     break;
452 case DeviceOutRequest | USB_REQ_SET_ADDRESS:
453     // wValue == urb->dev->devaddr
454     dev_dbg (hcd->self.controller, "root hub device address %d\n",
455             wValue);
456     break;
457
458 /* INTERFACE REQUESTS (no defined feature/status flags) */
459
460 /* ENDPOINT REQUESTS */
461
462 case EndpointRequest | USB_REQ_GET_STATUS:
463     // ENDPOINT_HALT flag
464     tbuf [0] = 0;
465     tbuf [1] = 0;
466     len = 2;
467     /* FALLTHROUGH */
468 case EndpointOutRequest | USB_REQ_CLEAR_FEATURE:
469 case EndpointOutRequest | USB_REQ_SET_FEATURE:
470     dev_dbg (hcd->self.controller, "no endpoint features yet\n");
471     break;
472
473 /* CLASS REQUESTS (and errors) */
474
475 default:
476     /* non-generic request */
477     switch (typeReq) {
478     case GetHubStatus:
479     case GetPortStatus:
480         len = 4;
481         break;
482     case GetHubDescriptor:
483         len = sizeof (struct usb_hub_descriptor);
484         break;
485     }
486     status = hcd->driver->hub_control (hcd,
487                                     typeReq, wValue, wIndex,
488                                     tbuf, wLength);
489     break;
490 error:
491     /* "protocol stall" on error */
492     status = -EPIPE;
493 }
494
495 if (status) {

```

```

496         len = 0;
497         if (status != -EPIPE) {
498             dev_dbg (hcd->self.controller,
499                     "CTRL: TypeReq=0x%x val=0x%x "
500                     "idx=0x%x len=%d ==> %d\n",
501                     typeReq, wValue, wIndex,
502                     wLength, status);
503         }
504     }
505     if (len) {
506         if (urb->transfer_buffer_length < len)
507             len = urb->transfer_buffer_length;
508         urb->actual_length = len;
509         // always USB_DIR_IN, toward host
510         memcpy (ubuf, bufp, len);
511
512         /* report whether RH hardware supports remote wakeup */
513         if (patch_wakeup &&
514             len > offsetof (struct usb_config_descriptor,
515                             bmAttributes))
516             ((struct usb_config_descriptor *)ubuf)->bmAttributes
517                 |= USB_CONFIG_ATT_WAKEUP;
518     }
519
520     /* any errors get returned through the urb completion */
521     local_irq_save (flags);
522     spin_lock (&urb->lock);
523     if (urb->status == -EINPROGRESS)
524         urb->status = status;
525     spin_unlock (&urb->lock);
526     usb_hcd_giveback_urb (hcd, urb);
527     local_irq_restore (flags);
528     return 0;
529 }

```

看到这样近 200 行的函数，真是有一种“叫天天不灵，叫地地不应”的感觉。不幸中的万幸，这个函数的结构还是很清晰的，自上而下地看过来就可以了。

对于控制传输，首先要获得它的 `setup_packet`，来自 `urb` 结构体，正如我们当初在 `usb-storage` 中看到的那样。这里把这个 `setup_packet` 赋给 `cmd` 指针。把其中的各个成员都给取出来，分别放在临时变量 `typeReq`，`wValue`，`wIndex`，`wLength` 中，然后来判断这个 `typeReq`。

如果是设备请求并且方向是 IN，而且是 `USB_REQ_GET_STATUS`，则设置 `len` 为 2。

如果是设备请求并且方向是 OUT，而且是 `USB_REQ_CLEAR_FEATURE`，则如何如何。

如果是设备请求并且方向是 OUT，而且是 `USB_REQ_SET_FEATURE`，则如何如何。

如果是设备请求并且方向是 IN，而且是 `USB_REQ_GET_CONFIGURATION`，则设置 `len` 为 1。

如果是设备请求并且方向是 OUT，而且是 `USB_REQ_SET_CONFIGURATION`，则什么也不做。

如果是设备请求并且方向是 IN，而且是 USB\_REQ\_GET\_DESCRIPTOR，则继续判断，wValue 值来决定究竟是要获得什么描述符。如果是 USB\_DT\_DEVICE，则说明要获得的是设备描述符，这正是咱们的上下文。（传递给 usb\_get\_descriptor 的第 2 个参数就是 USB\_DT\_DEVICE，传递给 usb\_control\_msg 的第 3 个参数正是 USB\_REQ\_GET\_DESCRIPTOR。）如果是 USB\_DT\_CONFIG，则说明要获得的是配置描述符；如果是 USB\_DT\_STRING，则说明要获得的是字符串描述符。实际上，对于 Root Hub 来说，这些东西都是一样的，在 drivers/usb/core/hcd.c 中都预先定义好了，usb2\_rh\_dev\_descriptor 是针对 USB 2.0 的；而 usb11\_rh\_dev\_descriptor 是针对 USB 1.1 的。HCI 驱动里面设置了 flags 的 HCD\_USB11。

```

116 #define KERNEL_REL      ((LINUX_VERSION_CODE >> 16) & 0x0ff)
117 #define KERNEL_VER      ((LINUX_VERSION_CODE >> 8) & 0x0ff)
118
119 /* usb 2.0 root hub device descriptor */
120 static const u8 usb2_rh_dev_descriptor [18] = {
121     0x12,          /* __u8 bLength; */
122     0x01,          /* __u8 bDescriptorType; Device */
123     0x00, 0x02, /* __le16 bcdUSB; v2.0 */
124
125     0x09,          /* __u8 bDeviceClass; HUB_CLASSCODE */
126     0x00,          /* __u8 bDeviceSubClass; */
127     0x01,          /* __u8 bDeviceProtocol; [ usb 2.0 single TT ]*/
128     0x40,          /* __u8 bMaxPacketSize0; 64 Bytes */
129
130     0x00, 0x00, /* __le16 idVendor; */
131     0x00, 0x00, /* __le16 idProduct; */
132     KERNEL_VER, KERNEL_REL, /* __le16 bcdDevice */
133
134     0x03,          /* __u8 iManufacturer; */
135     0x02,          /* __u8 iProduct; */
136     0x01,          /* __u8 iSerialNumber; */
137     0x01           /* __u8 bNumConfigurations; */
138 };
139
140 /* no usb 2.0 root hub "device qualifier" descriptor: one speed only */
141
142 /* usb 1.1 root hub device descriptor */
143 static const u8 usb11_rh_dev_descriptor [18] = {
144     0x12,          /* __u8 bLength; */
145     0x01,          /* __u8 bDescriptorType; Device */
146     0x10, 0x01, /* __le16 bcdUSB; v1.1 */
147
148     0x09,          /* __u8 bDeviceClass; HUB_CLASSCODE */
149     0x00,          /* __u8 bDeviceSubClass; */
150     0x00,          /* __u8 bDeviceProtocol; [ low/full speeds only ] */
151     0x40,          /* __u8 bMaxPacketSize0; 64 Bytes */
152
153     0x00, 0x00, /* __le16 idVendor; */
154     0x00, 0x00, /* __le16 idProduct; */
155     KERNEL_VER, KERNEL_REL, /* __le16 bcdDevice */
156
157     0x03,          /* __u8 iManufacturer; */
158     0x02,          /* __u8 iProduct; */
159     0x01,          /* __u8 iSerialNumber; */
160     0x01           /* __u8 bNumConfigurations; */

```

```

161 };
162
163 static const u8 fs_rh_config_descriptor [] = {
164
165     /* one configuration */
166     0x09,      /* __u8 bLength; */
167     0x02,      /* __u8 bDescriptorType; Configuration */
168     0x19, 0x00, /* __le16 wTotalLength; */
169     0x01,      /* __u8 bNumInterfaces; (1) */
170     0x01,      /* __u8 bConfigurationValue; */
171     0x00,      /* __u8 iConfiguration; */
172     0xc0,      /* __u8 bmAttributes;
173                    Bit 7: must be set,
174                    6: Self-powered,
175                    5: Remote wakeup,
176                    4..0: resvd */
177     0x00,      /* __u8 MaxPower; */
178
179     /* USB 1.1:
180     * USB 2.0, single TT organization (mandatory):
181     *     one interface, protocol 0
182     *
183     * USB 2.0, multiple TT organization (optional):
184     *     two interfaces, protocols 1 (like single TT)
185     *     and 2 (multiple TT mode) ... config is
186     *     sometimes settable
187     *     NOT IMPLEMENTED
188     */
189
190     /* one interface */
191     0x09,      /* __u8 if_bLength; */
192     0x04,      /* __u8 if_bDescriptorType; Interface */
193     0x00,      /* __u8 if_bInterfaceNumber; */
194     0x00,      /* __u8 if_bAlternateSetting; */
195     0x01,      /* __u8 if_bNumEndpoints; */
196     0x09,      /* __u8 if_bInterfaceClass; HUB_CLASSCODE */
197     0x00,      /* __u8 if_bInterfaceSubClass; */
198     0x00,      /* __u8 if_bInterfaceProtocol; [usb1.1 or single tt] */
199     0x00,      /* __u8 if_iInterface; */
200
201     /* one endpoint (status change endpoint) */
202     0x07,      /* __u8 ep_bLength; */
203     0x05,      /* __u8 ep_bDescriptorType; Endpoint */
204     0x81,      /* __u8 ep_bEndpointAddress; IN Endpoint 1 */
205     0x03,      /* __u8 ep_bmAttributes; Interrupt */
206     0x02, 0x00, /* __le16 ep_wMaxPacketSize; 1 + (MAX_ROOT_PORTS / 8) */
207     0xff,      /* __u8 ep_bInterval; (255 ms -- usb 2.0 spec) */
208 };
209
210 static const u8 hs_rh_config_descriptor [] = {
211
212     /* one configuration */
213     0x09,      /* __u8 bLength; */
214     0x02,      /* __u8 bDescriptorType; Configuration */
215     0x19, 0x00, /* __le16 wTotalLength; */
216     0x01,      /* __u8 bNumInterfaces; (1) */
217     0x01,      /* __u8 bConfigurationValue; */
218     0x00,      /* __u8 iConfiguration; */
219     0xc0,      /* __u8 bmAttributes;

```

```

225             Bit 7: must be set,
226             6: Self-powered,
227             5: Remote wakeup,
228             4..0: resvd */
229     0x00,      /* __u8 MaxPower; */
230
231     /* USB 1.1:
232     * USB 2.0, single TT organization (mandatory):
233     *     one interface, protocol 0
234     *
235     * USB 2.0, multiple TT organization (optional):
236     *     two interfaces, protocols 1 (like single TT)
237     *     and 2 (multiple TT mode) ... config is
238     *     sometimes settable
239     *     NOT IMPLEMENTED
240     */
241
242     /* one interface */
243     0x09,      /* __u8 if_bLength; */
244     0x04,      /* __u8 if_bDescriptorType; Interface */
245     0x00,      /* __u8 if_bInterfaceNumber; */
246     0x00,      /* __u8 if_bAlternateSetting; */
247     0x01,      /* __u8 if_bNumEndpoints; */
248     0x09,      /* __u8 if_bInterfaceClass; HUB_CLASSCODE */
249     0x00,      /* __u8 if_bInterfaceSubClass; */
250     0x00,      /* __u8 if_bInterfaceProtocol; [usb1.1 or single tt] */
251     0x00,      /* __u8 if_iInterface; */
252
253     /* one endpoint (status change endpoint) */
254     0x07,      /* __u8 ep_bLength; */
255     0x05,      /* __u8 ep_bDescriptorType; Endpoint */
256     0x81,      /* __u8 ep_bEndpointAddress; IN Endpoint 1 */
257     0x03,      /* __u8 ep_bmAttributes; Interrupt */
258             /* __le16 ep_wMaxPacketSize; 1 + (MAX_ROOT_PORTS / 8)
259             * see hub.c:hub_configure() for details. */
260     (USB_MAXCHILDREN + 1 + 7) / 8, 0x00,
261     0x0c      /* __u8 ep_bInterval; (256 ms -- usb 2.0 spec) */
262 };

```

如果是设备请求且方向为 IN，而且是 USB\_REQ\_GET\_INTERFACE，则设置 len 为 1。

如果是设备请求且方向为 OUT，而且是 USB\_REQ\_SET\_INTERFACE，则如何如何。

如果是设备请求且方向为 OUT，而且是 USB\_REQ\_SET\_ADDRESS，则如何如何。

如果是端点请求且方向为 IN，而且是 USB\_REQ\_GET\_STATUS，则如何如何。

如果是端点请求且方向为 OUT，而且是 USB\_REQ\_CLEAR\_FEATURE 或者 USB\_REQ\_SET\_FEATURE，则如何如何。

以上这些设置，统统是和 USB spec 中规定的东西相匹配的。

如果是 Hub 特定的类请求，而且是 GetHubStatus 或者是 GetPortStatus，则设置 len 为 4。

如果是 Hub 特定的类请求，而且是 GetHubDescriptor，则设置 len 为 usb\_hub\_descriptor 结



构体的大小。

最后对于 Hub 特定的类请求需要调用主机控制器驱动程序的 `hub_control` 函数，对于 `uhci_driver` 来说，这个指针被赋值为 `uhci_hub_control`，来自 `drivers/usb/host/uhci-hub.c`：

```

239 static int uhci_hub_control(struct usb_hcd *hcd, u16 typeReq, u16 wValue,
240                             u16 wIndex, char *buf, u16 wLength)
241 {
242     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
243     int status, lstatus, retval = 0, len = 0;
244     unsigned int port = wIndex - 1;
245     unsigned long port_addr = uhci->io_addr + USBPORTSC1 + 2 * port;
246     u16 wPortChange, wPortStatus;
247     unsigned long flags;
248
249     if (!test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags) || uhci->dead)
250         return -ETIMEDOUT;
251
252     spin_lock_irqsave(&uhci->lock, flags);
253     switch (typeReq) {
254
255     case GetHubStatus:
256         *(__le32 *)buf = cpu_to_le32(0);
257         OK(4); /* hub power */
258     case GetPortStatus:
259         if (port >= uhci->rh_numports)
260             goto err;
261
262         uhci_check_ports(uhci);
263         status = inw(port_addr);
264
265         /* Intel controllers report the OverCurrent bit active on.
266          * VIA controllers report it active off, so we'll adjust the
267          * bit value. (It's not standardized in the UHCI spec.)
268          */
269         if (to_pci_dev(hcd->self.controller)->vendor ==
270             PCI_VENDOR_ID_VIA)
271             status ^= USBPORTSC_OC;
272
273         /* UHCI doesn't support C_RESET (always false) */
274         wPortChange = lstatus = 0;
275         if (status & USBPORTSC_CSC)
276             wPortChange |= USB_PORT_STAT_C_CONNECTION;
277         if (status & USBPORTSC_PEC)
278             wPortChange |= USB_PORT_STAT_C_ENABLE;
279         if ((status & USBPORTSC_OCC) && !ignore_oc)
280             wPortChange |= USB_PORT_STAT_C_OVERCURRENT;
281
282         if (test_bit(port, &uhci->port_c_suspend)) {
283             wPortChange |= USB_PORT_STAT_C_SUSPEND;
284             lstatus |= 1;
285         }
286         if (test_bit(port, &uhci->resuming_ports))
287             lstatus |= 4;
288
289         /* UHCI has no power switching (always on) */
290         wPortStatus = USB_PORT_STAT_POWER;

```

```

291         if (status & USBPORTSC_CCS)
292             wPortStatus |= USB_PORT_STAT_CONNECTION;
293         if (status & USBPORTSC_PE) {
294             wPortStatus |= USB_PORT_STAT_ENABLE;
295             if (status & SUSPEND_BITS)
296                 wPortStatus |= USB_PORT_STAT_SUSPEND;
297         }
298         if (status & USBPORTSC_OC)
299             wPortStatus |= USB_PORT_STAT_OVERCURRENT;
300         if (status & USBPORTSC_PR)
301             wPortStatus |= USB_PORT_STAT_RESET;
302         if (status & USBPORTSC_LSDA)
303             wPortStatus |= USB_PORT_STAT_LOW_SPEED;
304
305         if (wPortChange)
306             dev_dbg(uhci_dev(uhci), "port %d portsc %04x,%02x\n",
307                     wIndex, status, lstatus);
308
309         *(__le16 *)buf = cpu_to_le16(wPortStatus);
310         *(__le16 *) (buf + 2) = cpu_to_le16(wPortChange);
311         OK(4);
312     case SetHubFeature:                /* We don't implement these */
313     case ClearHubFeature:
314         switch (wValue) {
315             case C_HUB_OVER_CURRENT:
316             case C_HUB_LOCAL_POWER:
317                 OK(0);
318             default:
319                 goto err;
320         }
321         break;
322     case SetPortFeature:
323         if (port >= uhci->rh_numports)
324             goto err;
325
326         switch (wValue) {
327             case USB_PORT_FEAT_SUSPEND:
328                 SET_RH_PORTSTAT(USBPORTSC_SUSP);
329                 OK(0);
330             case USB_PORT_FEAT_RESET:
331                 SET_RH_PORTSTAT(USBPORTSC_PR);
332
333                 /* Reset terminates Resume signalling */
334                 uhci_finish_suspend(uhci, port, port_addr);
335
336                 /* USB v2.0 7.1.7.5 */
337                 uhci->ports_timeout = jiffies + msecs_to_jiffies(50);
338                 OK(0);
339             case USB_PORT_FEAT_POWER:
340                 /* UHCI has no power switching */
341                 OK(0);
342             default:
343                 goto err;
344         }
345         break;
346     case ClearPortFeature:
347         if (port >= uhci->rh_numports)
348             goto err;
349

```

```

350         switch (wValue) {
351             case USB_PORT_FEAT_ENABLE:
352                 CLR_RH_PORTSTAT(USBPORTSC_PE);
353
354                 /* Disable terminates Resume signalling */
355                 uhci_finish_suspend(uhci, port, port_addr);
356                 OK(0);
357             case USB_PORT_FEAT_C_ENABLE:
358                 CLR_RH_PORTSTAT(USBPORTSC_PEC);
359                 OK(0);
360             case USB_PORT_FEAT_SUSPEND:
361                 if (!(inw(port_addr) & USBPORTSC_SUSP)) {
362
363                     /* Make certain the port isn't suspended */
364                     uhci_finish_suspend(uhci, port, port_addr);
365                 } else if (!test_and_set_bit(port,
366                                         &uhci->resuming_ports)) {
367                     SET_RH_PORTSTAT(USBPORTSC_RD);
368
369                     /* The controller won't allow RD to be set
370                      * if the port is disabled. When this happens
371                      * just skip the Resume signalling.
372                      */
373                     if (!(inw(port_addr) & USBPORTSC_RD))
374                         uhci_finish_suspend(uhci, port,
375                                             port_addr);
376                     else
377                         /* USB v2.0 7.1.7.7 */
378                         uhci->ports_timeout = jiffies +
379                                             msecs_to_jiffies(20);
380                 }
381                 OK(0);
382             case USB_PORT_FEAT_C_SUSPEND:
383                 clear_bit(port, &uhci->port_c_suspend);
384                 OK(0);
385             case USB_PORT_FEAT_POWER:
386                 /* UHCI has no power switching */
387                 goto err;
388             case USB_PORT_FEAT_C_CONNECTION:
389                 CLR_RH_PORTSTAT(USBPORTSC_CSC);
390                 OK(0);
391             case USB_PORT_FEAT_C_OVER_CURRENT:
392                 CLR_RH_PORTSTAT(USBPORTSC_OCC);
393                 OK(0);
394             case USB_PORT_FEAT_C_RESET:
395                 /* this driver won't report these */
396                 OK(0);
397             default:
398                 goto err;
399         }
400         break;
401     case GetHubDescriptor:
402         len = min_t(unsigned int, sizeof(root_hub_hub_des), wLength);
403         memcpy(buf, root_hub_hub_des, len);
404         if (len > 2)
405             buf[2] = uhci->rh_numports;
406         OK(len);
407     default:
408 err:

```

```

409         retval = -EPIPE;
410     }
411     spin_unlock_irqrestore(&uhci->lock, flags);
412
413     return retval;
414 }

```

249 行, struct usb\_hcd 结构体的成员 unsigned long flags, 当初在 usb\_add\_hcd 中调用 set\_bit 函数设置了这么一个 flag, HCD\_FLAG\_HW\_ACCESSIBLE, 基本上这个 flag 在我们的故事中是被设置了的。另外, struct uhci\_hcd 结构体有一个成员 unsigned int dead, 它如果为 1 就表明控制器“宕机”了。

然后用一个 switch 来处理 Hub 特定的类请求。OK 居然也是一个宏, 定义于 drivers/usb/host/uhci-hub.c:

```

78 #define OK(x)                len = (x); break

```

所以如果请求是 GetHubStatus, 则设置 len 为 4。

如果请求是 GetPortStatus, 则调用 uhci\_check\_ports, 然后读端口寄存器。USBPORTSC\_CSC 表示端口连接有变化, USBPORTSC\_PEC 表示端口 Enable 有变化。USBPORTSC\_OCC 表示 Over Current 有变化。struct uhci\_hcd 的两个成员, port\_c\_suspend 和 resuming\_ports 都是电源管理相关的。

但无论如何, 以上所做的这些都是为了获得: wPortStatus 和 wPortChange, 以此来响应 GetPortStatus 这个请求。

但是 SetPortFeature 就有事情要做了。wValue 表明具体是什么特征。

SET\_RH\_PORTSTAT 这个宏就是专门用于设置 Root Hub 的端口特征的。

```

80 #define CLR_RH_PORTSTAT(x) \
81     status = inw(port_addr); \
82     status &= ~(RWC_BITS|WZ_BITS); \
83     status &= ~(x); \
84     status |= RWC_BITS & (x); \
85     outw(status, port_addr)
86
87 #define SET_RH_PORTSTAT(x) \
88     status = inw(port_addr); \
89     status |= (x); \
90     status &= ~(RWC_BITS|WZ_BITS); \
91     outw(status, port_addr)

```

对于 USB\_PORT\_FEAT\_RESET, 还需要调用 uhci\_finish\_suspend。

如果是 USB\_PORT\_FEAT\_POWER, 则什么也不做, 因为 UHCI 不吃这一套。

如果请求是 ClearPortFeature, 基本上也是一样的做法。除了调用的宏变成了

## CLR\_RH\_PORTSTAT.

如果请求是 `GetHubDescriptor`，那就满足它。`root_hub_hub_des` 是早就在 `drivers/usb/host/uhci-hub.c` 中定义好的：

```
15 static __u8 root_hub_hub_des[] =
16 {
17     0x09,          /* __u8 bLength; */
18     0x29,          /* __u8 bDescriptorType; Hub-descriptor */
19     0x02,          /* __u8 bNbrPorts; */
20     0x0a,          /* __u16 wHubCharacteristics; */
21     0x00,          /* (per-port OC, no power switching) */
22     0x01,          /* __u8 bPwrOn2pwrGood; 2 ms */
23     0x00,          /* __u8 bHubContrCurrent; 0 mA */
24     0x00,          /* __u8 DeviceRemovable; *** 7 Ports max *** */
25     0xff           /* __u8 PortPwrCtrlMask; *** 7 ports max *** */
26 };
```

回到 `rh_call_control`，`switch` 结束了，下面判断 `status` 和 `len`。

然后调用 `usb_hcd_giveback_urb()`，来自 `drivers/usb/core/hcd.c`：

```
1385 void usb_hcd_giveback_urb (struct usb_hcd *hcd, struct urb *urb)
1386 {
1387     int at_root_hub;
1388
1389     at_root_hub = (urb->dev == hcd->self.root_hub);
1390     urb_unlink (urb);
1391
1392     /* lower level hcd code should use *_dma exclusively if the
1393      * host controller does DMA */
1394     if (hcd->self.uses_dma && !at_root_hub) {
1395         if (usb_pipecontrol (urb->pipe)
1396             && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
1397             dma_unmap_single (hcd->self.controller, urb->setup_dma,
1398                             sizeof (struct usb_ctrlrequest),
1399                             DMA_TO_DEVICE);
1400         if (urb->transfer_buffer_length != 0
1401             && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
1402             dma_unmap_single (hcd->self.controller,
1403                             urb->transfer_dma,
1404                             urb->transfer_buffer_length,
1405                             usb_pipein (urb->pipe)
1406                                 ? DMA_FROM_DEVICE
1407                                 : DMA_TO_DEVICE);
1408     }
1409
1410     usbmon_urb_complete (&hcd->self, urb);
1411     /* pass ownership to the completion handler */
1412     urb->complete (urb);
1413     atomic_dec (&urb->use_count);
1414     if (unlikely (urb->reject))
1415         wake_up (&usb_kill_urb_queue);
1416     usb_put_urb (urb);
1417 }
1418 EXPORT_SYMBOL (usb_hcd_giveback_urb);
```

这里最重要最有意义的一行当然就是 1412 行，调用 `urb` 的 `complete` 函数，这正是在 `usb-storage` 里期待的那个函数。从此 `rh_call_control` 函数也该返回了，以后设备驱动又获得了控制权。事实上令人欣喜的是对于 Root Hub，1394 行开始的这一段 `if` 是不会被执行的，因为 `at_root_hub` 显然是为真。不过就算这段要执行也没什么可怕的，无非就是把之前为 `urb` 建立的 DMA 映射给取消掉。而另一方面，对于 Root Hub 来说，`complete` 函数基本上是什么也不做，只不过是让咱们再次回到 `usb_start_wait_urb` 去，而控制传输需要的数据也已经 `copy` 到 `urb->transfer_buffer` 中去了。至此，Root Hub 的控制传输就算结束了，即我们的 `usb_get_device_descriptor` 函数取得了空前绝后的圆满成功。

## 17. 非 Root Hub 的批量传输

看完了控制传输，再来看批量传输，Root hub 没有批量传输，所以只需要关注非 Root Hub。

当然还是从 `usb_submit_urb()` 开始。和控制传输一样，可以直接跳到 `usb_hcd_submit_urb()`。由于我们在 `start_rh()` 中设置了 `hcd->state` 为 `HC_STATE_RUNNING`，所以这里 `list_add_tail` 会被执行，本 `urb` 会被加入到 `ep` 的 `urb_list` 队列中去。

然后还是老套路，`driver->urb_enqueue` 会被执行，即又一次进入了 `uhci_urb_enqueue`。没啥好说的，`uhci_alloc_urb_priv` 和 `uhci_alloc_qh` 会被再次执行以申请 `urbp` 和 `QH`，但这次 `uhci_submit_control` 不会被调用了，取而代之的是 `uhci_submit_bulk()`。这个函数来自 `drivers/usb/host/uhci-qc.c`：

```

1043 static int uhci_submit_bulk(struct uhci_hcd *uhci, struct urb *urb,
1044                             struct uhci_qh *qh)
1045 {
1046     int ret;
1047
1048     /* Can't have low-speed bulk transfers */
1049     if (urb->dev->speed == USB_SPEED_LOW)
1050         return -EINVAL;
1051
1052     if (qh->state != QH_STATE_ACTIVE)
1053         qh->skel = SKEL_BULK;
1054     ret = uhci_submit_common(uhci, urb, qh);
1055     if (ret == 0)
1056         uhci_add_fsbr(uhci, urb);
1057     return ret;
1058 }

```

又是一个很“赤裸裸”的函数，除了设置 `qh->skel` 为 `SKEL_BULK` 以外，就是调用 `uhci_submit_common` 了，而这个函数也是我们今后将在中断传输中调用的。因为批量传输和中断传输一样，就是一个阶段，直接传递数据就可以了，不用那么多废话。如果成功返回的话在

调用 `uhci_add_fsbr` 把 `urbp->fsbr` 设置为 1。我们来看一下 `uhci_submit_common()`，这是一个公共的函数，批量传输和中断传输都会调用它，来自 `drivers/usb/host/uhci-q.c`：

```

927 static int uhci_submit_common(struct uhci_hcd *uhci, struct urb *urb,
928                               struct uhci_qh *qh)
929 {
930     struct uhci_td *td;
931     unsigned long destination, status;
932     int maxsize = le16_to_cpu(qh->hep->desc.wMaxPacketSize);
933     int len = urb->transfer_buffer_length;
934     dma_addr_t data = urb->transfer_dma;
935     __le32 *plink;
936     struct urb_priv *urbp = urb->hcpriv;
937     unsigned int toggle;
938
939     if (len < 0)
940         return -EINVAL;
941
942     /* The "pipe" thing contains the destination in bits 8--18 */
943     destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
944     toggle = usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
945                            usb_pipeout(urb->pipe));
946
947     /* 3 errors, dummy TD remains inactive */
948     status = uhci_maxerr(3);
949     if (urb->dev->speed == USB_SPEED_LOW)
950         status |= TD_CTRL_LS;
951     if (usb_pipein(urb->pipe))
952         status |= TD_CTRL_SPD;
953
954     /*
955      * Build the DATA TDs
956      */
957     plink = NULL;
958     td = qh->dummy_td;
959     do { /* Allow zero length packets */
960         int pktsize = maxsize;
961
962         if (len <= pktsize) { /* The last packet */
963             pktsize = len;
964             if (!(urb->transfer_flags & URB_SHORT_NOT_OK))
965                 status &= ~TD_CTRL_SPD;
966         }
967
968         if (plink) {
969             td = uhci_alloc_td(uhci);
970             if (!td)
971                 goto nomem;
972             *plink = LINK_TO_TD(td);
973         }
974         uhci_add_td_to_urbp(td, urbp);
975         uhci_fill_td(td, status,
976                     destination | uhci_explen(pktsize) |
977                     (toggle << TD_TOKEN_TOGGLE_SHIFT),
978                     data);
979         plink = &td->link;
980         status |= TD_CTRL_ACTIVE;
981     } while (len > 0);

```

```

982         data += pktsize;
983         len -= maxsize;
984         toggle ^= 1;
985     } while (len > 0);
986
987     /*
988     * URB_ZERO_PACKET means adding a 0-length packet, if direction
989     * is OUT and the transfer_length was an exact multiple of maxsize,
990     * hence (len = transfer_length - N * maxsize) == 0
991     * however, if transfer_length == 0, the zero packet was already
992     * prepared above.
993     */
994     if ((urb->transfer_flags & URB_ZERO_PACKET) &&
995         usb_pipeout(urb->pipe) && len == 0 &&
996         urb->transfer_buffer_length > 0) {
997         td = uhci_alloc_td(uhci);
998         if (!td)
999             goto nomem;
1000         *plink = LINK_TO_TD(td);
1001
1002         uhci_add_td_to_urbp(td, urbp);
1003         uhci_fill_td(td, status,
1004                     destination | uhci_explen(0) |
1005                     (toggle << TD_TOKEN_TOGGLE_SHIFT),
1006                     data);
1007         plink = &td->link;
1008
1009         toggle ^= 1;
1010     }
1011
1012     /* Set the interrupt-on-completion flag on the last packet.
1013     * A more-or-less typical 4 KB URB (= size of one memory page)
1014     * will require about 3 ms to transfer; that's a little on the
1015     * fast side but not enough to justify delaying an interrupt
1016     * more than 2 or 3 URBs, so we will ignore the URB_NO_INTERRUPT
1017     * flag setting. */
1018     td->status |= __constant_cpu_to_le32(TD_CTRL_IOC);
1019
1020     /*
1021     * Build the new dummy TD and activate the old one
1022     */
1023     td = uhci_alloc_td(uhci);
1024     if (!td)
1025         goto nomem;
1026     *plink = LINK_TO_TD(td);
1027
1028     uhci_fill_td(td, 0, USB_PID_OUT | uhci_explen(0), 0);
1029     wmb();
1030     qh->dummy_td->status |= __constant_cpu_to_le32(TD_CTRL_ACTIVE);
1031     qh->dummy_td = td;
1032
1033     usb_settoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1034                  usb_pipeout(urb->pipe), toggle);
1035     return 0;
1036
1037 nomem:
1038     /* Remove the dummy TD from the td_list so it doesn't get freed */
1039     uhci_remove_td_from_urbp(qh->dummy_td);
1040     return -ENOMEM;

```



几经曲折之后我终于看明白了这个函数，虽然这个函数很“无耻”，但是它却给了我们一丝亲切的感觉，我们曾经熟悉的 `urb`，曾经熟悉的 `transfer_buffer_length` 以及 `transfer_dma` 又一次映入了我们的眼帘。这里我们看到它们俩被赋给了 `len` 和 `data`。

932 行，令 `maxsize` 等于端点描述符里记录的 `wMaxPacketSize`，即包的最大“size”。

接下来又是一堆的赋值：

第 1 个，`destination`，`urb->pipe` 由几个部分组成，这里的两个宏无非就是提取其中的 `destination`，它们都来自 `drivers/usb/host/uhci-hcd.h`：

```
7 #define usb_packetid(pipe) \
    (usb_pipein(pipe) ? USB_PID_IN : USB_PID_OUT)
8 #define PIPE_DEVEP_MASK      0x0007ff00
```

显然，`PIPE_DEVEP_MASK` 就是用来获取 bit8 到 bit18，而 `usb_packetid` 就是为了获得传输的方向，IN 还是 OUT，`usb_pipein` 就是获取 `pipe` 的 bit7。

第 2 个，`toggle`，`usb_gettoggle` 就是获得这个 `toggle` 位。

第 3 个，`status`，等式右边的 `uhci_maxerr` 来自 `drivers/usb/host/uhci-hcd.h`：

```
203 #define uhci_maxerr(err)      ((err) << TD_CTRL_C_ERR_SHIFT)
```

在同一个文件中还定义了这样一些宏：

```
184 #define TD_CTRL_SPD            (1 << 29)      /* Short Packet Detect */
185 #define TD_CTRL_C_ERR_MASK    (3 << 27)      /* Error Counter bits */
186 #define TD_CTRL_C_ERR_SHIFT    27
187 #define TD_CTRL_LS            (1 << 26)      /* Low Speed Device */
188 #define TD_CTRL_IOS            (1 << 25)      /* Isochronous Select */
189 #define TD_CTRL_IOC            (1 << 24)      /* Interrupt on Complete */
190 #define TD_CTRL_ACTIVE        (1 << 23)      /* TD Active */
191 #define TD_CTRL_STALLED        (1 << 22)      /* TD Stalled */
192 #define TD_CTRL_DBUFFERR        (1 << 21)      /* Data Buffer Error */
193 #define TD_CTRL_BABBLE        (1 << 20)      /* Babble Detected */
194 #define TD_CTRL_NAK            (1 << 19)      /* NAK Received */
195 #define TD_CTRL_CRCTIMEO        (1 << 18)      /* CRC/Time Out Error */
196 #define TD_CTRL_BITSTUFF        (1 << 17)      /* Bit Stuff Error */
197 #define TD_CTRL_ACTLEN_MASK    0x7FF /* actual length, encoded as n - 1 */
198
199 #define TD_CTRL_ANY_ERROR      (TD_CTRL_STALLED | TD_CTRL_DBUFFERR | \
200                                TD_CTRL_BABBLE | TD_CTRL_CRCTIMEO | \
201                                TD_CTRL_BITSTUFF)
```

这些宏看似很莫名其妙，其实是有来历的，UHCI spec 中对 TD 有明确的描述，硬件上来看，它一共有四个双字（DWORD）。这其中第二个双字被称为 TD CONTROL AND STATUS，就是专门记录控制和状态信息的，一个双字就是 32 个 bits，bit0 到 bit31。而其中咱们这里的 `TD_CTRL_C_ERR_MASK` 就是为了提取 bit28 和 bit27 的。spec 中说，这两位是一个计数器，

记录的是这个 TD 在执行过程中被探测到出现错误的次数，比如一开始设置它为 3，那么它每次出现错误就减 1，三次错误之后这个计数器就从 1 变成了 0，于是主机控制器就会把这个 TD 设置为 inactive，即给这个 TD 宣判“死刑”。咱们这里设置的就是 3 次。

接下来几行还是设置这个 status，status 的 bit26 标志着这个设备是低速设备还是全速设备。如果为 1 则表示是低速设备，为 0 则表示是全速设备。bit29 表示 Short Packet Detect (SPD)，其含义为，如果这一位为 1，则当一个包是输入包，并且成功完成了传输，但是实际长度比最大长度要短，则这个 TD 将会被标为 inactive。如果是输出包，则这一位没有任何意义。所以这里判断的是这个管道是不是输入管道。另外，如果传输出现了错误，则这一位也没有任何意义，汇报 SPD 的前提是数据必须成功地被传输了。

在做好这些前奏工作之后，957 行开始干正经事了。

如果传输的长度比 pktsize，或者说小于 maxsize，则说明这个包是最后一个包了。URB\_SHORT\_NOT\_OK 是 urb 的 transfer\_flags 中众多标志位的一个，如果设置了这一个 flag 就表明 short 包是不能够接受的。反之则说明确实是一个短包，这种情况就把 status 中 SPD 这一位给清掉。

接着看，plink 一开始被设置为 NULL。所以第一次进入循环的话就直接执行 974 行，uhci\_add\_td\_to\_urbp()，

然后调用 uhci\_fill\_td 函数，咱们已经讲过了，无非就是设置一个 TD 的 status, token 和 buffer 这三个成员。

设置了 TD 之后，令 plink 等于 td->link，TD 的 link 也是 UHCI spec 明确规定的 4 个 DWORD 之一，被称为 Link Pointer，物理上，正是它把各个 TD 给连接起来的。

设置好这些之后，再把 status 中 bit23 给设置为 1，这一位如果为 1，则表示激活这个传输了。TD\_CTRL\_ACTIVE 这一位用来表征 TD 是一个待执行的活跃交互，主机控制器驱动在调度一个交互请求时将这一位设成 1，而硬件（主机控制器）在完成了一次交互之后，或者成功，或者彻底失败，就将这一位改成 0。这样驱动程序只要扫描各个 uhci\_td 数据结构，发现某个 uhci\_td 数据结构的 TD\_CTRL\_ACTIVE 位变成了 0，就说明这个交互已经完成。

最后增加 data，减小 len，并且把 toggle 位置反。如果数据还没传输完，就开始下一轮的循环。

第二次循环的区别在于 plink 这时候已经有值了，所以这次 969 行 uhci\_alloc\_td 会被执行，这次就将申请一个 TD。然后让 plink 里的内容赋为这个 TD 的 DMA 地址，这样就把这个 TD 和之前的 TD 给连接了起来。而其他事情则和第一次循环时一样。

不过有人问了这么一个问题，这里貌似有两个队列，一个是 td->list，一个是 td->link，这

是什么原因？我们看到 `struct uhci_td` 中有 `__le32 link` 和 `struct list_head list`，后者就是一个经典的队列头，而前者是一个链接指针，实际上它们构成了两个队列，或者说两个链表，前者使用的物理地址，后者使用的是虚拟地址。因为 USB 主机控制器显然不认识虚拟地址。所以我们要让 USB 主机控制器能够顺着各个 TD 来执行，就得为它准备一个物理地址链接起来的队列，但是同时，从软件角度来说，要保证 CPU 能够访问各个 TD，则又必须以虚拟地址的方式建立一个队列，从而使得 CPU 可以对 `uhci_td` 数据结构进行常规的队列操作。在我们的故事中出现了两个队列。`uhci_submit_common` 函数结束后，各个 TD 就组成了一个 QH。

这个循环结束之后，主机控制器的驱动工作就算完成了，我们知道处理器的基本职责是取指令和执行指令，类似地，UHCI 主机控制器的基本职责就是取 TD 和执行 TD，这里因为 TD 也建好了，也连入该连接的地方了，剩下的具体执行就是硬件的事情了。

其实建立 TD 队列的过程是很简单的，反反复复的就是在调用三个函数：`uhci_alloc_td`，`uhci_add_td_to_urbp`，`uhci_fill_td`。其意图很明显，基本上就是三步走，申请 TD，将其加入大部队，填充好。其中每一次调用了 `uhci_alloc_td` 之后都要判断是否申请成功，如果不成功就直接 `goto nomem`。

然后还有一些细节的工作，994 行，判断 `urb` 的另一个 `transfer_flags`，`URB_ZERO_PACKET` 是否设置了，如果设置了，并且传输方向是输出，`len` 等于 0，需要传输的数据长度是大于 0 的（这说明最初 `len` 并不是 0，而现在是 0，即说明 `transfer_buffer_length` 的长度恰好等于整数个 `maxsze`）。这个 `flag` 的含义是这个传输最后需要有一个零长度的包。对于这种情况，没啥好说的，申请一个 TD 连接、填充好，然后把 `toggle` 位置反即可。

1018 行，设置 `status` 的 `bit24`（`interrupt on Complete(IOC)`）。这一位如果为 1，则表示主机控制器会在这个 TD 执行的 `Frame` 结束时触发中断。当初咱们在 `uhci_submit_control` 中也给状态阶段的 TD 设置了这一位。

接下来的这一小段代码基本上就是处理那个 `dummy_td`。当年在 `uhci_alloc_qh` 中曾经刻意给 `qh->dummy_td` 给申请了空间。TD 是用来结束一个队列的，或者说它表征队列的结束。

这里结束之后，这个函数就结束了，返回 0。只有刚才申请 TD 时失败了才会跳到下面去执行 `uhci_remove_td_from_urbp()`，把 `dummy_td` 从 `td_list` 中删除。这个函数也是很简单的，来自 `drivers/usb/host/uhci-q.c`：

```
151 static void uhci_remove_td_from_urbp(struct uhci_td *td)
152 {
153     list_del_init(&td->list);
154 }
```

然后，`uhci_submit_common` 结束之后回到 `uhci_submit_bulk`，并进而回到 `uhci_urb_enqueue` 中，而剩下的代码和控制传输就一样了，无须多说。这样，传说中的批量传输就这么被轻松搞定了。同样，在数据真的执行完之后，会执行 `urb` 的 `complete` 函数，控制权会转移给设备驱动。

这一切听上去都很完美，似乎天衣无缝，可问题是，不管是之前说的控制传输还是现在说的批量传输，`urb->complete` 究竟是被谁调用的？前面在讲 Root Hub 时咱们看到了 `usb_hcd_giveback_urb` 被调用，而它会调用 `urb->complete`。那么对于非 Root hub 呢？

还记得咱们注册了中断函数吧？中断函数不会“吃闲饭”，为控制传输和批量传输中的最后一个 TD 设置了 IOC，于是该 TD 完成之后的那个 Frame 结束时分，主机控制器会向 CPU 发送中断，于是中断函数会被调用。

## 18. 传说中的中断服务程序（ISR）

想当初在 `usb_add_hcd` 中使用 `request_irq` 注册了中断函数，每注册一个函数都是为了日后能够利用该函数，当初注册了 `usb_hcd_irq`，这会儿就该调用这个函数了。这个函数来自 `drivers/usb/core/hcd.c`：

```
1431 irqreturn_t usb_hcd_irq (int irq, void *__hcd)
1432 {
1433 struct usb_hcd      *hcd = __hcd;
1434     int              start = hcd->state;
1435
1436     if (unlikely(start == HC_STATE_HALT ||
1437         !test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags)))
1438         return IRQ_NONE;
1439     if (hcd->driver->irq (hcd) == IRQ_NONE)
1440         return IRQ_NONE;
1441
1442     set_bit(HCD_FLAG_SAW_IRQ, &hcd->flags);
1443
1444     if (unlikely(hcd->state == HC_STATE_HALT))
1445         usb_hc_died (hcd);
1446     return IRQ_HANDLED;
1447 }
```

对于 UHCI 来说，`driver->irq` 就是 `uhci_irq()` 函数。来自 `drivers/usb/host/uhci-hcd.c`：

```
377 static irqreturn_t uhci_irq(struct usb_hcd *hcd)
378 {
379     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
380     unsigned short status;
381     unsigned long flags;
382
383     /*
384     * Read the interrupt status, and write it back to clear the
385     * interrupt cause. Contrary to the UHCI specification, the
386     * "HC Halted" status bit is persistent: it is RO, not R/WC.
387     */
388     status = inw(uhci->io_addr + USBSTS);
389     if (!(status & ~USBSTS_HCH)) /* shared interrupt, not mine */
390         return IRQ_NONE;
```

```

391     outw(status, uhci->io_addr + USBSTS);          /* Clear it */
392
393     if (status & ~(USBSTS_USBINT | USBSTS_ERROR | USBSTS_RD)) {
394         if (status & USBSTS_HSE)
395             dev_err(uhci_dev(uhci), "host system error, "
396                     "PCI proble ms?\n");
397         if (status & USBSTS_HCPE)
398             dev_err(uhci_dev(uhci), "host controller process "
399                     "error, something bad happened!\n");
400         if (status & USBSTS_HCH) {
401             spin_lock_irqsave(&uhci->lock, flags);
402             if (uhci->rh_state >= UHCI_RH_RUNNING) {
403                 dev_err(uhci_dev(uhci),
404                         "host controller halted, "
405                         "very bad!\n");
406                 if (debug > 1 && errbuf) {
407                     /* Print the schedule for debugging */
408                     uhci_sprint_schedule(uhci,
409                                           errbuf, ERRBUF_LEN);
410                     lprintk(errbuf);
411                 }
412                 uhci_hc_died(uhci);
413
414                 /* Force a callback in case there are
415                  * pending unlinks */
416                 mod_timer(&hcd->rh_timer, jiffies);
417             }
418             spin_unlock_irqrestore(&uhci->lock, flags);
419         }
420     }
421
422     if (status & USBSTS_RD)
423         usb_hcd_poll_rh_status(hcd);
424     else {
425         spin_lock_irqsave(&uhci->lock, flags);
426         uhci_scan_schedule(uhci);
427         spin_unlock_irqrestore(&uhci->lock, flags);
428     }
429
430     return IRQ_HANDLED;
431 }

```

USBSTS 就是 UHCI 的状态寄存器，而 USBSTS\_USBINT 标志状态寄存器的 bit0，按照 UHCI spec 的规定，bit0 对应于 IOC。USBSTS\_ERROR 对应于 bit 1，这一位如果为 1，表示传输出现了错误，USBSTS\_RD 则对应于 bit2，RD 就是 Resume Detect 的意思，主机控制器在收到 resume 的信号时会把这一位设置为 1。所以很快我们就知道我们应该关注的就是 426 这么一行代码，即 `uhci_scan_schedule` 这个最熟悉的陌生人。

当我们再一次踏入 `uhci_scan_schedule` 时，曾经那段被略过的 while 循环现在就不得不面对了，`uhci_advance_check` 会被调用，它来自 `drivers/usb/host/uhci-q.c`：

```

1636 static int uhci_advance_check(struct uhci_hcd *uhci, struct uhci_qh *qh)
1637 {
1638     struct urb_priv *urbp = NULL;
1639     struct uhci_td *td;

```

```

1640     int ret = 1;
1641     unsigned status;
1642
1643     if (qh->type == USB_ENDPOINT_XFER_ISOC)
1644         goto done;
1645
1646     /* Treat an UNLINKING queue as though it hasn't advanced.
1647     * This is okay because reactivation will treat it as though
1648     * it has advanced, and if it is going to become IDLE then
1649     * this doesn't matter anyway. Furthermore it's possible
1650     * for an UNLINKING queue not to have any URBs at all, or
1651     * for its first URB not to have any TDs (if it was dequeued
1652     * just as it completed). So it's not easy in any case to
1653     * test whether such queues have advanced. */
1654     if (qh->state != QH_STATE_ACTIVE) {
1655         urbp = NULL;
1656         status = 0;
1657
1658     } else {
1659         urbp = list_entry(qh->queue.next, struct urb_priv, node);
1660         td = list_entry(urbp->td_list.next, struct uhci_td, list);
1661         status = td_status(td);
1662         if (!(status & TD_CTRL_ACTIVE)) {
1663
1664             /* We're okay, the queue has advanced */
1665             qh->wait_expired = 0;
1666             qh->advance_jiffies = jiffies;
1667             goto done;
1668         }
1669         ret = 0;
1670     }
1671
1672     /* The queue hasn't advanced; check for timeout */
1673     if (qh->wait_expired)
1674         goto done;
1675
1676     if (time_after(jiffies, qh->advance_jiffies + QH_WAIT_TIMEOUT)) {
1677
1678         /* Detect the Intel bug and work around it */
1679         if (qh->post_td && qh_element(qh) == LINK_TO_TD(qh->post_td)) {
1680             qh->element = qh->post_td->link;
1681             qh->advance_jiffies = jiffies;
1682             ret = 1;
1683             goto done;
1684         }
1685
1686         qh->wait_expired = 1;
1687
1688         /* If the current URB wants FSBR, unlink it temporarily
1689         * so that we can safely set the next TD to interrupt on
1690         * completion. That way we'll know as soon as the queue
1691         * starts moving again. */
1692         if (urbp && urbp->fsbr && !(status & TD_CTRL_IOC))
1693             uhci_unlink_qh(uhci, qh);
1694
1695     } else {
1696         /* Unmoving but not-yet-expired queues keep FSBR alive */
1697         if (urbp)
1698             uhci_urbp_wants_fsbr(uhci, urbp);

```

```

1699     }
1700
1701 done:
1702     return ret;
1703 }

```

从 `urbp` 中的 `td_list` 里面取出一个 TD，读取它的状态，最初设置了 `TD_CTRL_ACTIVE`，如果一个 TD 被执行完了，主机控制器会把它的 `TD_CTRL_ACTIVE` 给取消掉。所以这里 1662 行判断，如果已经没有了 `TD_CTRL_ACTIVE`，说明这个 TD 已经被执行完了，于是执行 `goto` 语句跳出去，从而 `uhci_advance_check` 函数就返回值了，对于这种情况，返回值为 1。`uhci_advance_check` 顾名思义，就是检查咱们的队列有没有前进，如果一个 TD 从 `ACTIVE` 变成了非 `ACTIVE`，这就说明队列前进了，因为主机控制器只有执行完一个 TD 才会把一个 TD 的 `ACTIVE` 取消，然后它就会前进去获取下一个 QH 或者 TD。

而如果 `uhci_advance_check` 返回了 1，那么接下来 `uhci_scan_qh` 会被调用，它来自 `drivers/usb/host/uhci-q.c`：

```

1536 static void uhci_scan_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
1537 {
1538     struct urb_priv *urbp;
1539     struct urb *urb;
1540     int status;
1541
1542     while (!list_empty(&qh->queue)) {
1543         urbp = list_entry(qh->queue.next, struct urb_priv, node);
1544         urb = urbp->urb;
1545
1546         if (qh->type == USB_ENDPOINT_XFER_ISOC)
1547             status = uhci_result_isochronous(uhci, urb);
1548         else
1549             status = uhci_result_common(uhci, urb);
1550         if (status == -EINPROGRESS)
1551             break;
1552
1553         spin_lock(&urb->lock);
1554         if (urb->status == -EINPROGRESS) /* Not dequeued */
1555             urb->status = status;
1556         else
1557             status = ECONNRESET; /* Not -ECONNRESET */
1558         spin_unlock(&urb->lock);
1559
1560         /* Dequeued but completed URBs can't be given back unless
1561          * the QH is stopped or has finished unlinking. */
1562         if (status == ECONNRESET) {
1563             if (QH_FINISHED_UNLINKING(qh))
1564                 qh->is_stopped = 1;
1565             else if (!qh->is_stopped)
1566                 return;
1567         }
1568
1569         uhci_giveback_urb(uhci, qh, urb);
1570         if (status < 0 && qh->type != USB_ENDPOINT_XFER_ISOC)
1571             break;
1572     }

```

```

1573
1574 /* If the QH is neither stopped nor finished unlinking (normal case),
1575  * our work here is done. */
1576 if (QH_FINISHED_UNLINKING(qh))
1577     qh->is_stopped = 1;
1578 else if (!qh->is_stopped)
1579     return;
1580
1581 /* Otherwise give back each of the dequeued URBs */
1582 restart:
1583 list_for_each_entry(urbp, &qh->queue, node) {
1584     urb = urbp->urb;
1585     if (urb->status != -EINPROGRESS) {
1586
1587         /* Fix up the TD links and save the toggles for
1588          * non-Isochronous queues. For Isochronous queues,
1589          * test for too-recent dequeues. */
1590         if (!uhci_cleanup_queue(uhci, qh, urb)) {
1591             qh->is_stopped = 0;
1592             return;
1593         }
1594         uhci_giveback_urb(uhci, qh, urb);
1595         goto restart;
1596     }
1597 }
1598 qh->is_stopped = 0;
1599
1600 /* There are no more dequeued URBs. If there are still URBs on the
1601  * queue, the QH can now be re-activated. */
1602 if (!list_empty(&qh->queue)) {
1603     if (qh->needs_fixup)
1604         uhci_fixup_toggles(qh, 0);
1605
1606     /* If the first URB on the queue wants FSBR but its time
1607      * limit has expired, set the next TD to interrupt on
1608      * completion before reactivating the QH. */
1609     urbp = list_entry(qh->queue.next, struct urb_priv, node);
1610     if (urbp->fsbr && qh->wait_expired) {
1611         struct uhci_td *td = list_entry(urbp->td_list.next,
1612                                         struct uhci_td, list);
1613
1614         td->status |= __cpu_to_le32(TD_CTRL_IOC);
1615     }
1616     uhci_activate_qh(uhci, qh);
1617 }
1618
1619 /* The queue is empty. The QH can become idle if it is fully
1620  * unlinked. */
1621 else if (QH_FINISHED_UNLINKING(qh))
1622     uhci_make_qh_idle(uhci, qh);
1623 }
1624 }

```

可以看到,不管是控制传输还是批量传输,下一个被调用的函数都是 uhci\_result\_common(), 来自 drivers/usb/host/uhci-qc:

```

1151 static int uhci_result_common(struct uhci_hcd *uhci, struct urb *urb)
1152 {

```



```

1153     struct urb_priv *urbp = urb->hcpriv;
1154     struct uhci_qh *qh = urbp->qh;
1155     struct uhci_td *td, *tmp;
1156     unsigned status;
1157     int ret = 0;
1158
1159     list_for_each_entry_safe(td, tmp, &urbp->td_list, list) {
1160         unsigned int ctrlstat;
1161         int len;
1162
1163         ctrlstat = td_status(td);
1164         status = uhci_status_bits(ctrlstat);
1165         if (status & TD_CTRL_ACTIVE)
1166             return -EINPROGRESS;
1167
1168         len = uhci_actual_length(ctrlstat);
1169         urb->actual_length += len;
1170
1171         if (status) {
1172             ret = uhci_map_status(status,
1173                                   uhci_packetout(td_token(td)));
1174             if ((debug == 1 && ret != -EPIPE) || debug > 1) {
1175                 /* Some debugging code */
1176                 dev_dbg(&urb->dev->dev,
1177                         "%s: failed with status %x\n",
1178                         __FUNCTION__, status);
1179
1180                 if (debug > 1 && errbuf) {
1181                     /* Print the chain for debugging */
1182                     uhci_show_qh(uhci, urbp->qh, errbuf,
1183                                 ERRBUF_LEN, 0);
1184                     lprintk(errbuf);
1185                 }
1186             }
1187
1188             } else if (len < uhci_expected_length(td_token(td))) {
1189
1190                 /* We received a short packet */
1191                 if (urb->transfer_flags & URB_SHORT_NOT_OK)
1192                     ret = -EREMOTEIO;
1193
1194                 /* Fixup needed only if this isn't the URB's last TD */
1195                 else if (&td->list != urbp->td_list.prev)
1196                     ret = 1;
1197             }
1198
1199             uhci_remove_td_from_urbp(td);
1200             if (qh->post_td)
1201                 uhci_free_td(uhci, qh->post_td);
1202             qh->post_td = td;
1203
1204             if (ret != 0)
1205                 goto err;
1206         }
1207     }
1208     return ret;
1209 err:
1210     if (ret < 0) {
1211         /* In case a control transfer gets an error

```

```

1212      * during the setup stage */
1213      urb->actual_length = max(urb->actual_length, 0);
1214
1215      /* Note that the queue has stopped and save
1216       * the next toggle value */
1217      qh->element = UHCI_PTR_TERM;
1218      qh->is_stopped = 1;
1219      qh->needs_fixup = (qh->type != USB_ENDPOINT_XFER_CONTROL);
1220      qh->initial_toggle = uhci_toggle(td_token(td)) ^
1221                          (ret == -EREMOTEIO);
1222
1223  } else          /* Short packet received */
1224      ret = uhci_fixup_short_transfer(uhci, qh, urbp);
1225  return ret;
1226 }

```

首先 `list_for_each_entry_safe` 就相当于传说中的 `list_for_each_entry`，其作用都是遍历 `urbp` 的 `td_list`，一个一个 TD 地处理。

1163 行，`td_status` 是一个很简单的宏，来自 `drivers/usb/host/uhci-hcd.h`：

```

262 static inline u32 td_status(struct uhci_td *td) {
263     __le32 status = td->status;
264
265     barrier();
266     return le32_to_cpu(status);
267 }

```

其实就是获取 `struct uhci_td` 结构体指针的 `status` 成员。

而 `uhci_status_bits` 亦是来自同一个文件中的宏：

```

204 #define uhci_status_bits(ctrl_sts)    ((ctrl_sts) & 0xF60000)

```

要看懂这个宏需要参考下面的图 3.14.1。

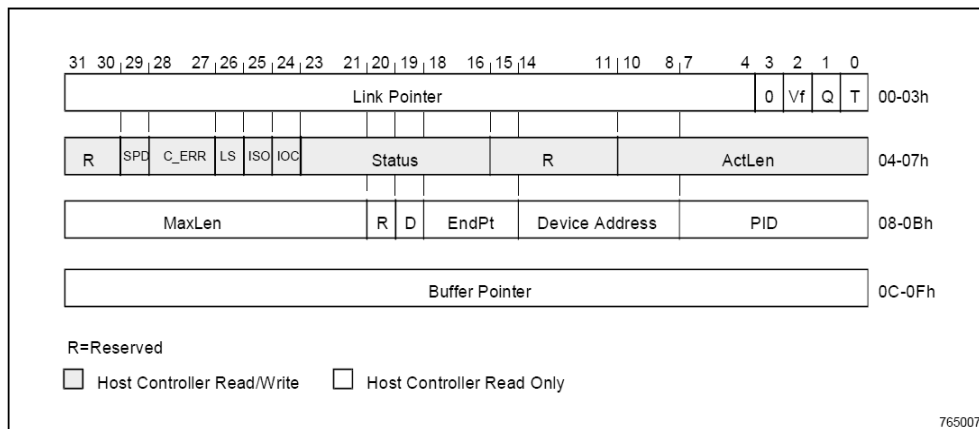


图 3.14.1 TD 结构定义

这就是 UHCI spec 中对 TD 的结构体的定义，我们注意到它有 4 个 DWORD，而 `uhci_td` 中的成员 `status` 实际上指的是这里的 04-07h 这整个双字，请注意这幅图中 04-07h 这个双字中，bit16 到 bit23 那一段被称为 Status，即这几位表示状态，`uhci_status_bits` 则是为了获得这几个 bits，把 `ctrl_sts` 和 0xF60000 相与得到 bit17 到 bit23，因为 UHCI spec 中规定了 bit16 是保留位，意义不大。

这其中，bit 23 被称为 Active，其实它就是 we 一直说的那个 `TD_CTRL_ACTIVE`。如果这一位还设置了那么就说明这个 TD 还是活的，不要去碰它。如果没有设置，那么继续往下走。

下一个宏是 `uhci_actual_length`，依然来自 `drivers/usb/host/uhci-hcd.h`：

```
205 #define uhci_actual_length(ctrl_sts) (((ctrl_sts) + 1) & \
206                                     TD_CTRL_ACTLEN_MASK) /* 1-based */
```

这里 `TD_CTRL_ACTLEN_MASK` 是 0x7FF，我们注意到 TD 的定义中，在 04~07h 中，bit0 到 bit10 这 11 个 bits 被称之为 `ActLen`，这个 field 是由主机控制器来写的，表示实际传输了多少个 Bytes，它被以  $n-1$  的方式进行编码，所以这里解码就要加 1。在 `usb-storage` 那个故事中看到的 `urb->actual_length` 就是这么计算出来的，即每次处理一个 TD 就加上 `len`。

顺便提一下，我们注意到在 `uhci_submit_control` 中设置了 `urb->actual_length` 为 -8，实际上用 `urb->actual_length` 小于 0 来表示控制传输的 Setup 阶段没能取得成功，至于它具体是负多少并不重要，取为负 8 只是图个吉利。

如果一切正常的话，`status` 实际上应该是 0，不为 0 就表示出错了。1171 这一段就是为错误打印一些调试信息。

1188，如果虽然没有啥异常状态，但是 `len` 比期望值要小，那么首先判断是不是在 `urb` 的 `transfer_flags` 中设置了 `URB_SHORT_NOT_OK`，如果设置了，那就返回汇报错误。如果没有设置，继续判断，查看这个 TD 是不是咱们整个队伍中最后一个 TD，否则就有问题，设置返回值 1。

1199 行，既然 TD 完成了使命，那么就可以“过河拆桥、卸磨杀驴”了。

1200 行，第一次见 `qh->post_td`，它当然是空的。如果不为空就调用 `uhci_free_td` 来释放它。`struct uhci_qh` 结构体中的成员 `post_td` 是用来记录刚刚完成了的那个 TD。它的赋值恰恰就是在 1202 这一行，即令 `qh->post_td` 等于现在这个 TD，因为这个 TD 就是刚刚完成的 TD。

正常的话，应该返回 0。如果不正常，那就跳到 1209 下面去。

如果 `ret` 小于 0，则需要对 QH 的一些成员进行赋值。

如果 `ret` 不小于 0，实际上就是对应于刚才那个 `ret` 为 1 的情况，即传输长度小于预期长度，这种情况就调用 `uhci_fixup_short_transfer()` 这个专门为此而设计的函数。来自

drivers/usb/host/uhci-q.c:

```

1104 static int uhci_fixup_short_transfer(struct uhci_hcd *uhci,
1105                                     struct uhci_qh *qh, struct urb_priv *urbp)
1106 {
1107     struct uhci_td *td;
1108     struct list_head *tmp;
1109     int ret;
1110
1111     td = list_entry(urbp->td_list.prev, struct uhci_td, list);
1112     if (qh->type == USB_ENDPOINT_XFER_CONTROL) {
1113
1114         /* When a control transfer is short, we have to restart
1115          * the queue at the status stage transaction, which is
1116          * the last TD. */
1117         WARN_ON(list_empty(&urbp->td_list));
1118         qh->element = LINK_TO_TD(td);
1119         tmp = td->list.prev;
1120         ret = -EINPROGRESS;
1121     } else {
1122
1123         /* When a bulk/interrupt transfer is short, we have to
1124          * fix up the toggles of the following URBs on the queue
1125          * before restarting the queue at the next URB. */
1126         qh->initial_toggle = uhci_toggle(td_token(qh->post_td)) ^ 1;
1127         uhci_fixup_toggles(qh, 1);
1128
1129         if (list_empty(&urbp->td_list))
1130             td = qh->post_td;
1131         qh->element = td->link;
1132         tmp = urbp->td_list.prev;
1133         ret = 0;
1134     }
1135 }
1136
1137 /* Remove all the TDs we skipped over, from tmp back to the start */
1138 while (tmp != &urbp->td_list) {
1139     td = list_entry(tmp, struct uhci_td, list);
1140     tmp = tmp->prev;
1141
1142     uhci_remove_td_from_urbp(td);
1143     uhci_free_td(uhci, td);
1144 }
1145 return ret;
1146 }

```

这里对于控制传输和对于批量传输有着不同的处理方法。

如果是控制传输，那么令 `tmp` 等于本 `urb` 的 `td_list` 中的倒数第 2 个 TD，然后一个一个往前走，见一个删一个，并且把 `ret` 设置为 `-EINPROGRESS` 然后返回 `ret`，这样做的后果就是留下了最后一个 TD，而其他 TD 统统撤了。而对于控制传输，我们知道其最后一个 TD 就是状态阶段的 TD。

而对于批量传输或者中断传输的做法是从最后一个 TD 开始往前走，全都删除。  
`uhci_fixup_toggles()`来自 `drivers/usb/host/uhci-q.c`:

```

377 static void uhci_fixup_toggles(struct uhci_qh *qh, int skip_first)
378 {
379     struct urb_priv *urbp = NULL;
380     struct uhci_td *td;
381     unsigned int toggle = qh->initial_toggle;
382     unsigned int pipe;
383
384     /* Fixups for a short transfer start with the second URB in the
385      * queue (the short URB is the first). */
386     if (skip_first)
387         urbp = list_entry(qh->queue.next, struct urb_priv, node);
388
389     /* When starting with the first URB, if the QH element pointer is
390      * still valid then we know the URB's toggles are okay. */
391     else if (qh_element(qh) != UHCI_PTR_TERM)
392         toggle = 2;
393
394     /* Fix up the toggle for the URBs in the queue. Normally this
395      * loop won't run more than once: When an error or short transfer
396      * occurs, the queue usually gets emptied. */
397     urbp = list_prepare_entry(urbp, &qh->queue, node);
398     list_for_each_entry_continue(urbp, &qh->queue, node) {
399
400         /* If the first TD has the right toggle value, we don't
401          * need to change any toggles in this URB */
402         td = list_entry(urbp->td_list.next, struct uhci_td, list);
403         if (toggle > 1 || uhci_toggle(td_token(td)) == toggle) {
404             td = list_entry(urbp->td_list.prev, struct uhci_td,
405                             list);
406             toggle = uhci_toggle(td_token(td)) ^ 1;
407
408             /* Otherwise all the toggles in the URB have to be switched */
409             else {
410                 list_for_each_entry(td, &urbp->td_list, list) {
411                     td->token ^= __constant_cpu_to_le32(
412                                     TD_TOKEN_TOGGLE);
413                     toggle ^= 1;
414                 }
415             }
416         }
417
418         wmb();
419         pipe = list_entry(qh->queue.next, struct urb_priv, node)->urb->pipe;
420         usb_settoggle(qh->udev, usb_pipeendpoint(pipe),
421                     usb_pipeout(pipe), toggle);
422         qh->needs_fixup = 0;
423     }

```

哇噻，这一段美妙的队列操作，足以让我等看得眼花缭乱头晕目眩了。

看这个函数之前，注意两点：

第一，在调用 `uhci_fixup_toggles` 之前的那句话，`qh->initial_toggle` 被赋了值，而且还有就是 `post_td` 的 `toggle` 位取反。

第二，咱们传递进来的第 2 个参数是 1，即 `skip_first` 是 1，因此 387 行会被执行，`urbp` 是

QH 的 queue 队列中的第 2 个。因为第 1 个必然是刚才处理的那个，即那个出现短包问题的 urb。

然后 397, 398 行从第 2 个 urbp 开始遍历 QH 的 queue 队列。首先是获得 urbp 里的第 1 个 TD。注意到 toggle 要么为 1 要么为 0。（除非 skip\_first 为 0，执行了 392 行，那么 toggle 将等于 2。）如果这个 TD 的 toggle 位和 qh->initial\_toggle 相同，即它和那个 post\_td 的 toggle 相反，那么 TD 是正确的，直接让 TD 走到 td\_list 的最后一个元素去，再把 toggle 置为反。

反之，如果 TD 的 toggle 和 qh->initial\_toggle 不同，即它和之前那个 post\_td 的 toggle 相同，那么说明整个 urb 中的所有的 TD 的 toggle 位都反了，都得翻一次。

最后调用 usb\_settoggle 来设置一次，设置 qh->needs\_fixup 为 0。

显然，这么说谁都会，关键是得理解，也许现在是时候去理解一下 USB 世界里的同步问题了。USB spec 中是为了实现同步，定义了 Data0 和 Data1 这两种序列位，如果发送人员要发送多个包给接收者，则给每个包编上号，让 Data0 和 Data1 间隔着发送出去。发送方和接收方都维护着一张绪列位，在一次传输开始之前，发送方和接收方的这个序列位必须同步，或者说相同，即要么同为 Data0，要么同为 Data1，这种机制称之为 Data Toggle Synchronization。

举例来说，假设一开始双方都是 Data0，当接收方成功地接收到了一个包，会把自己的同步位翻转，即所谓的 toggle，它变成了 Data1，然后发送一个 ACK 给发送方，告诉对方，已成功地接收到了你的包，而发送方在接收到这个 ACK 之后，也会翻转自己的同步位，于是也跟着变成了 Data1。下一个包也是一样的做法。所以我们看到 uhci\_submit\_common() 函数中没填充一个 TD，就翻转一次 toggle 位，即 984 行那个 “toggle^=1”。同样在 uhci\_submit\_control() 中也能看到对于 toggle 的处理。回过头去看 uhci\_submit\_control() 中 879 行，来看一下我们是如何为控制传输设置 toggle 位的。

首先是 Setup 阶段，让 toggle 位为 0。（A SETUP always uses a DATA0 PID for the data field of the SETUP transaction. ---USB spec 2.0, 8.5.3）

其次是数据阶段，在填充每一个 TD 之前翻转 toggle 位，即 850 行那个 destination^=TD\_TOKEN\_TOGGLE，第一次翻转之后 toggle 位是 Data1。

最后是状态阶段，879 行，为状态阶段的 toggle 位设置为 Data1，这是依据 USB spec 中规定的来设置的。（A Status stage is delineated by a change in direction of data flow from the previous stage and always uses a DATA1 PID. ---USB spec 2.0, 8.5.3）

那么为何在 uhci\_submit\_common 中调用了 usb\_gettoggle() 和 usb\_settoggle，而 uhci\_submit\_control 中没有调用呢？struct usb\_device 这个结构体有这么一个成员 toggle[]。

```
336 struct usb_device {
337     int          devnum;          /* Address on USB bus */
338     char         devpath [16];    /* Use in messages: /port/port/... */
339     enum usb_device_state state; /* configured, not attached, etc */
```

```

340     enum usb_device_speed    speed; /* high/full/low (or error) */
341
342     struct usb_tt    *tt; /* low/full speed dev, highspeed hub */
343     int              ttport; /* device port on that tt hub */
344
345     unsigned int toggle[2]; /* one bit for each endpoint
346                             * ([0] = IN, [1] = OUT) */
347

```

`toggle` 数组的第 1 个元素是针对 IN 类型端点的，第 2 个元素是针对 OUT 类型端点的，每个端点都在这张表里占有一个 bit。于是咱们就可以用它来记录端点的 `toggle`，以保证传输的同步，但是，实际上在这个故事里，真正使用这个数组的只有两种端点，即批量端点和中断端点，另外两种端点并不需要这个数组。首先，等时端点是不需要使用 `toggle bits` 来进行同步的，这是 USB spec 中规定的。Data Toggle 同步对等时传输没有意义。其次，控制传输的 `toggle` 位的 Setup 阶段总是 Data0，数据阶段总是从 Data1 开始，Status 阶段总是 Data1。

USB spec 已经为控制传输规定好了，必须遵守它，所以就没有必要另外使用这个数组来记录端点的 `toggle` 位了。这就是为什么操作这个 `toggle` 数组的两个函数 `usb_gettoggle/usb_settoggle` 不会出现在提交控制 urb 的函数 `uhci_submit_control` 中。而对于批量传输和中断传输，恰恰是因为每次在设置好一个 urb 的各个 TD 之后调用 `usb_settoggle` 来设置这个 `toggle`，下一次为新 urb 的第一个 TD 设置 `toggle` 位时才可以直接调用 `usb_gettoggle`。这样就保证了前一个 urb 的 TD 的 `toggle` 位和后一个 urb 的 TD 的 `toggle` 位刚好相反，即所谓的交叉顺序，以保证了和设备内部的 `toggle` 位相同步。

了解了这些 `toggle` 位的设置之后，再来看我们的这段代码，来看一下这个 `uhci_fixup_toggles` 究竟是怎么“fixup”的。根据前面看到的对 `qh->initial_toggle` 的赋值可以知道，`initial_toggle` 实际上就是接收到 short 包的那个 TD 的 `toggle` 位取反，即 `post_td` 的 `toggle` 取反（函数 `uhci_fixup_short_transfer` 中 1127 行），而 403 行所比较的就是第 2 个 urb 的第 1 个 TD 的 token 是否和现在这个一样，如果不一样，就把该 urb 的所有的 TD 翻转一下，如果一样，则说明没有问题，但无论哪种情况，都要记录 `toggle` 本身，因为注意到在 420 行还调用了 `usb_settoggle` 来设置了该管道的 `toggle` 位的。

那么为什么说如果一样就说明没有问题呢？我们知道，主机控制器处理的 TD 总是 QH 中的第 1 个 TD，当然其所属的 urb 也一定是 QH 的第 1 个 urb，而且该 TD 的 `toggle` 位是和端点同步的，假设它们之前都是 Data0，那么现在该 TD 结束之后，端点那边的 `toggle` 位就该变成了 Data1。

另一方面，根据 UHCI spec，我们知道，如果一个 urb 的 TD 被检测到了短包，则该 urb 剩下的 TD 就不会被处理了，而下一个 urb 的第 1 个 TD 的 `toggle` 得和现在这个 urb 的被处理的 TD 的 `toggle` 相反就说明它的 `toggle` 位也是 Data1，即它是和端点同步的。这样就可以理直气壮地重新开启下一个 urb。反之，如果第 1 个 TD 和端点的 `toggle` 位相反，就把整个队列的所有 TD 都给反一下，这个工程不可谓不浩大，但是没有办法，谁叫设备不争气发送出这种短包来

呢？这就叫成长的代价。

另外提一下，和 `uhci_submit_common()` 函数一样，也可以理解为什么在 `uhci_fixup_toggles` 最后，即 420 行，会再次调用 `usb_settoggle` 了。注意一下，403 至 415 这一段，`toggle` 的两种赋值：第一种，由于整个“队伍”是出于正确的同步状况，所以不用改任何一个 TD 的 `toggle` 位，404 行直接让 `td` 等于本 `urb` 队列中的最后一个 TD，然后 `toggle` 是它的 `toggle` 位取反。而对于整个队伍都得翻转的情况，看到 411 让每一个 `td` 进行翻转，而 413 行 `toggle` 也跟着一次次翻转，以保证 `toggle` 最终等于最后一个 `td` 的 `toggle` 位的翻转。

最后再来看一下 `TD_CTRL_SPD` 这个 flag 的使用。这个 flag 对应于 TD 那 4 个双字中的第二个双字中的 `bit29`，在 UHCI spec 中关于 `bit` 是这么介绍的：

```
Short Packet Detect (SPD). 1=Enable. 0=Disable. When a packet has this bit set to 1 and the packet:
1. is an input packet;
2. is in a queue; and
3. successfully completes with an actual length less than the maximum length;
then the TD is marked inactive, the Queue Header is not updated and the USBINT status bit (Status Register) is set at the end of the frame. In addition, if the interrupt is enabled, the interrupt will be sent at the end of the frame.
Note that any error (e.g., babble or FIFO error) prevents the short packet from being reported. The behavior is undefined when this bit is set with output packets or packets outside of queues.
```

所以，对于 IN 方向的数据包，如果设置了这个 flag，那么主机控制器读到一个短包之后，它就会触发中断。因此注意到 `uhci_submit_common` 函数中，951 行和 952 行，就对 IN 管道设置了这个 flag。即对于接下来的每一个数据包，都会检测一下看是否收到了短包，是的话就及时发送中断向上级汇报。而 965 行又取消掉这个 flag 了，因为这是最后一个包，最后一个包当然是有可能是短包的。同样，在 `uhci_submit_control` 中也是如法炮制，835 行设置了 `TD_CTRL_SPD`，即保证数据阶段能够准确地汇报“险情”，而 881 行又取消掉，因为这已经是状态阶段了，最后一个包当然是允许短包的。

最后注意到，`uhci_fixup_toggles` 最后一行设置了 `qh->needs_fixup` 为 0。稍后会看到对这个变量是否为 0 的判断，目前这个上下文当然就是 0。

回到 `uhci_fixup_short_transfer` 来，一个需要解释的问题是，为何要设置 `qh->element`。正如上面从 UHCI spec 中摘取过来的那段对 SPD 的解释中所说的，当遇到短包时，QH 不会被更新“update”，这也是为什么一个 TD 出现了短包下一个 TD 就不会被执行的原因。所以这里咱们就需要手工的“update”这个 QH。对于控制传输，QH 的 `element` 指向了状态传输的 TD，因为要让状态阶段重新执行一次，就算是短包也得汇报一下，所以最后返回的是 `-EINPROGRESS`。而对于批量/中断传输，TD 是本 `urbp` 的 `td_list` 中最后一个 TD（看 1111 行的赋值）。`element` 指向了该 TD 的 `link` 指针，也就是指向了下一个 `urb`，所以最后返回的是 0。

到这里就很明白，`uhci_fixup_short_transfer()` 中 1138 行 1144 这一段 `while` 循环的意义了。



把那个有问题的 urb 的前面那些 TD 统统删掉，把内存也释放掉。

至此，我们结束了 `uhci_fixup_short_transfer()`。因而，`uhci_result_common` 也就结束了。回到了 `uhci_scan_qh` 中，仍然在 QH 中按照 urb 一个一个地循环。如果 status 是 `-EINPROGRESS`，则结束循环，继续执行该 urb。

没什么故障的话，`urb->status` 应该还是 `-EINPROGRESS`，这是最初提交 urb 时设置的。于是设置 `urb->status` 为 status，这就是执行之后的结果。

最后 1569 行，既然 status 不是 `-EINPROGRESS`，那么 `uhci_giveback_urb` 被调用。

```

1485 static void uhci_giveback_urb(struct uhci_hcd *uhci, struct uhci_qh *qh,
1486                               struct urb *urb)
1487 __releases(uhci->lock)
1488 __acquires(uhci->lock)
1489 {
1490     struct urb_priv *urbp = (struct urb_priv *) urb->hcpriv;
1491
1492     /* When giving back the first URB in an Isochronous queue,
1493      * reinitialize the QH's iso-related members for the next URB. */
1494     if (qh->type == USB_ENDPOINT_XFER_ISOC &&
1495         urbp->node.prev == &qh->queue &&
1496         urbp->node.next != &qh->queue) {
1497         struct urb *nurb = list_entry(urbp->node.next,
1498                                       struct urb_priv, node)->urb;
1499
1500         qh->iso_packet_desc = &nurb->iso_frame_desc[0];
1501         qh->iso_frame = nurb->start_frame;
1502         qh->iso_status = 0;
1503     }
1504
1505     /* Take the URB off the QH's queue. If the queue is now empty,
1506      * this is a perfect time for a toggle fixup. */
1507     list_del_init(&urbp->node);
1508     if (list_empty(&qh->queue) && qh->needs_fixup) {
1509         usb_settoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1510                      usb_pipeout(urb->pipe), qh->initial_toggle);
1511         qh->needs_fixup = 0;
1512     }
1513
1514     uhci_free_urb_priv(uhci, urbp);
1515
1516     spin_unlock(&uhci->lock);
1517     usb_hcd_giveback_urb(uhci_to_hcd(uhci), urb);
1518     spin_lock(&uhci->lock);
1519
1520     /* If the queue is now empty, we can unlink the QH and give up its
1521      * reserved bandwidth. */
1522     if (list_empty(&qh->queue)) {
1523         uhci_unlink_qh(uhci, qh);
1524         if (qh->bandwidth_reserved)
1525             uhci_release_bandwidth(uhci, qh);
1526     }
1527 }
```

首先 1494 行这一段 if 是针对等时传输的，暂时略过。

然后把这个 urbp 从 QH 的队伍中删除掉。如果队列因此就空了，并且 needs\_fixup 设置为了 1。那就调用 usb\_settoggle。不过上下文里 needs\_fixup 是 0，所以暂不管。把 urbp 的各个 TD 全部给删除，把 TD 的内存给释放掉，把 urbp 本身的内存释放掉。

接下来调用 usb\_hcd\_giveback\_urb 把控制权交回给设备驱动程序。这个函数已经不再陌生了。

最后，如果 QH 整个队伍已经空了，那么就调用 uhci\_unlink\_qh 把 QH 给撤掉。这个函数来自 drivers/usb/host/uhci-q.h:

```
555 static void uhci_unlink_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
556 {
557     if (qh->state == QH_STATE_UNLINKING)
558         return;
559     WARN_ON(qh->state != QH_STATE_ACTIVE || !qh->udev);
560     qh->state = QH_STATE_UNLINKING;
561
562     /* Unlink the QH from the schedule and record when we did it */
563     if (qh->skel == SKEL_ISO)
564         ;
565     else if (qh->skel < SKEL_ASYNC)
566         unlink_interrupt(uhci, qh);
567     else
568         unlink_async(uhci, qh);
569
570     uhci_get_current_frame_number(uhci);
571     qh->unlink_frame = uhci->frame_number;
572
573     /* Force an interrupt so we know when the QH is fully unlinked */
574     if (list_empty(&uhci->skel_unlink_qh->node))
575         uhci_set_next_interrupt(uhci);
576
577     /* Move the QH from its old list to the end of the unlinking list */
578     if (qh == uhci->next_qh)
579         uhci->next_qh = list_entry(qh->node.next, struct uhci_qh,
580                                 node);
581     list_move_tail(&qh->node, &uhci->skel_unlink_qh->node);
582 }
```

对于批量传输或者控制传输，要调用的是 unlink\_async()，依然是来自 drivers/usb/host/uhci-q.c:

```
537 static void unlink_async(struct uhci_hcd *uhci, struct uhci_qh *qh)
538 {
539     struct uhci_qh *pqh;
540     __le32 link_to_next_qh = qh->link;
541
542     pqh = list_entry(qh->node.prev, struct uhci_qh, node);
543     pqh->link = link_to_next_qh;
544
545     /* If this was the old first FSBR QH, link the terminating skeleton
546      * QH to the next (new first FSBR) QH. */
547     if (pqh->skel < SKEL_FSBR && qh->skel >= SKEL_FSBR)
```

```

548         uhci->skel_term_qh->link = link_to_next_qh;
549         mb();
550     }

```

“打江山难而毁江山容易”，这一句话从 `link_async` 和 `unlink_async` 这两个函数对比一下就会明白其真义。540 行、542 行、543 行的结果就是经典的删除队列节点的操作。让 `pqh` 等于 `qh` 的前一个节点，然后让 `pqh` 的 `link` 等于原来 `qh` 的 `link`，这样 `qh` 就没有利用价值了。

547 行这个 `if` 也不难理解，如果刚才的 `qh` 是第 1 个 FSBR 的 `qh`，那么就令 `skel_term_qh` 的 `link` 指向下一个 `qh`，因为前面说过，`skel_term_qh` 总是要被设置为第 1 个 FSBR `qh`。

调用 `uhci_get_current_frame_number` 获得当前的 Frame，记录在 `unlink_frame` 中。

调用 `uhci_set_next_interrupt`，来自 `drivers/usb/host/uhci-q.c`：

```

28 static void uhci_set_next_interrupt(struct uhci_hcd *uhci)
29 {
30     if (uhci->is_stopped)
31         mod_timer(&uhci->hcd(uhci)->rh_timer, jiffies);
32     uhci->term_td->status |= cpu_to_le32(TD_CTRL_IOC);
33 }

```

这个函数的行为显然是和 `uhci_clear_next_interrupt` 相反的，等于开启中断。

如果这个 `qh` 是 `uhci->next_qh`，那么就让 `next_qh` 顺延至下一个 QH。

最后把刚才 `unlink` 的这个 QH 插入到另外一支队伍中去，这支队伍就是 `uhci->skel_unlink_qh`，所有被 `unlink` 的 QH 都会被招入这支“革命队伍”中去。很显然这是一支“无产阶级革命队伍”，因为进来的 `qh` 都是一无所有的。

`uhci_giveback_urb` 结束了，回到 `uhci_scan_qh` 中。`uhci_cleanup_queue` 被调用，来自 `drivers/usb/host/uhci-q.c`：

```

319 static int uhci_cleanup_queue(struct uhci_hcd *uhci, struct uhci_qh *qh,
320                             struct urb *urb)
321 {
322     struct urb_priv *urbp = urb->hcpriv;
323     struct uhci_td *td;
324     int ret = 1;
325
326     /* Isochronous pipes don't use toggles and their TD link pointers
327      * get adjusted during uhci_urb_dequeue(). But since their queues
328      * cannot truly be stopped, we have to watch out for dequeues
329      * occurring after the nominal unlink frame. */
330     if (qh->type == USB_ENDPOINT_XFER_ISOC) {
331         ret = (uhci->frame_number + uhci->is_stopped !=
332              qh->unlink_frame);
333         goto done;
334     }
335
336     /* If the URB isn't first on its queue, adjust the link pointer
337      * of the last TD in the previous URB. The toggle doesn't need

```

```

338      * to be saved since this URB can't be executing yet. */
339      if (qh->queue.next != &urbp->node) {
340          struct urb_priv *purbp;
341          struct uhci_td *ptd;
342
343          purbp = list_entry(urbp->node.prev, struct urb_priv, node);
344          WARN_ON(list_empty(&purbp->td_list));
345          ptd = list_entry(purbp->td_list.prev, struct uhci_td,
346                          list);
347          td = list_entry(urbp->td_list.prev, struct uhci_td,
348                          list);
349          ptd->link = td->link;
350          goto done;
351      }
352
353      /* If the QH element pointer is UHCI_PTR_TERM then then currently
354       * executing URB has already been unlinked, so this one isn't it. */
355      if (qh_element(qh) == UHCI_PTR_TERM)
356          goto done;
357      qh->element = UHCI_PTR_TERM;
358
359      /* Control pipes don't have to worry about toggles */
360      if (qh->type == USB_ENDPOINT_XFER_CONTROL)
361          goto done;
362
363      /* Save the next toggle value */
364      WARN_ON(list_empty(&urbp->td_list));
365      td = list_entry(urbp->td_list.next, struct uhci_td, list);
366      qh->needs_fixup = 1;
367      qh->initial_toggle = uhci_toggle(td_token(td));
368
369  done:
370      return ret;
371 }

```

最后，uhci\_make\_qh\_idle 被调用，来自 drivers/usb/host/uhci-q.c:

```

590 static void uhci_make_qh_idle(struct uhci_hcd *uhci, struct uhci_qh *qh)
591 {
592     WARN_ON(qh->state == QH_STATE_ACTIVE);
593
594     if (qh == uhci->next_qh)
595         uhci->next_qh = list_entry(qh->node.next, struct uhci_qh,
596                                   node);
597     list_move(&qh->node, &uhci->idle_qh_list);
598     qh->state = QH_STATE_IDLE;
599
600     /* Now that the QH is idle, its post_td isn't being used */
601     if (qh->post_td) {
602         uhci_free_td(uhci, qh->post_td);
603         qh->post_td = NULL;
604     }
605
606     /* If anyone is waiting for a QH to become idle, wake them up */
607     if (uhci->num_waiting)
608         wake_up_all(&uhci->waitqh);
609 }

```

目的就一个：设置 `qh->state` 为 `QH_STATE_IDLE`。

`uhci_make_qh_idle` 结束之后，`uhci_scan_qh` 也就结束了，回到了 `uhci_scan_schedule` 中。

最后判断 `uhci->skel_unlink_qh` 领衔的“队伍”是否为空，如果为空，就调用 `uhci_clear_next_interrupt` 清中断，否则又有 QH 加入了这支“队伍”，就调用 `uhci_set_next_interrupt` 去产生下一次中断，从而再次把 `qh->state` 设置为 `QH_STATE_IDLE`。于是 `uhci_scan_schedule` 也结束了。而 `uhci_irq` 也就结束了。

## 19. Root Hub 的中断传输

中断传输和等时传输无疑要比之前的那两种传输复杂些，至少它们具有周期性。我们看代码时看不懂也不用灰心，歌手林志炫的成名曲《单身情歌》也就是把自己看代码的亲身经历唱了出来：“为了看代码孤军奋斗，早就吃够了看代码的苦，在代码中失落的人到处有，而我只是其中一个”。

和前面那个控制传输一样，中断传输的代码也分为两部分：一个是针对 Root Hub 的，这部分相当简单；另一个是针对非 Root Hub 的，这一部分明显复杂许多。先来看 Root Hub 的。跟踪 `urb` 的入口多少年也不会变，依然是 `usb_submit_urb`。

还记得在 Hub 驱动中讲的那个 `hub_probe` 吗？在 Hub 驱动的探测过程中，最终会提交一个 `urb`，即中断 `urb`。那么来看调用 `usb_submit_urb()` 来 submit 这个 `urb` 之后究竟会发生什么？

但是与之前控制传输批量传输不同的是，之前是“凌波微步”来到了最后一行 `usb_hcd_submit_urb`，而现在在这一行之前还有几行是需要关注的。

首先注意到 243 行对临时变量 `temp` 赋了值，看到它被赋值为 `usb_pipetype(pipe)`，我相信即使是普通人也知道，从此以后 `temp` 就是管道类型。

于是“飞”到 338 行，这里有一个 `switch`。看到这里你明白当初在讲控制传输和 Bulk 传输时为何跳过了这一段了吧，没错，这里只有两个 `case`，即 `PIPE_ISOCHRONOUS` 和 `PIPE_INTERRUPT`，这两个 `case` 就是等时管道和中断管道。而控制传输和 Bulk 传输根本不在这一个选择的考虑范畴之内。所以当时我们很幸福地“飘”了过去。但现在不行了。实际上这里对于等时传输和对于中断传输处理方法是一样的。

首先判断 `urb->interval`，在 Hub 驱动中已经讲过它的作用，它当然不能小于等于 0。

其次根据设备是高速还是全速低速，再一次设置 `interval`。当时在 Hub 驱动中也说过，对

于高速设备和全速低速设备，这个 `interval` 的单位是不一样的。前者的单位是微帧，后者的单位是帧，所以这里对它们有不同的处理方法，但是总的来说，可以看到 `temp` 无论如何会是 2 的整数次方，所以 372 行这么一赋值的效果是，如果你的期待值是介于 2 的  $n$  次方和 2 的  $n+1$  次方之间，那么就把它设置成 2 的  $n$  次方。因为最终设置成 2 的整数次方对我们来说软件上便于实现，而硬件上来说也无所谓，因为 USB spec 中也是允许的，比如，USB spec 2.0 5.7.4 中有这么一段：

“The period provided by the system may be shorter than that desired by the device up to the shortest period defined by the USB (125  $\mu$ s microframe or 1 ms frame). The client software and device can depend only on the fact that the host will ensure that the time duration between two transaction attempts with the endpoint will be no longer than the desired period.”

现在进入 `usb_hcd_submit_urb`。而对于 Root Hub，我们又再一次进入了 `rh_urb_enqueue`；对于中断传输，我们进入了 `rh_queue_status`，这个函数来自 `drivers/usb/core/hcd.c`：

```

600 static int rh_queue_status (struct usb_hcd *hcd, struct urb *urb)
601 {
602     int          retval;
603     unsigned long flags;
604     int          len = 1 + (urb->dev->maxchild / 8);
605
606     spin_lock_irqsave (&hcd_root_hub_lock, flags);
607     if (urb->status != -EINPROGRESS)      /* already unlinked */
608         retval = urb->status;
609     else if (hcd->status_urb || urb->transfer_buffer_length < len) {
610         dev_dbg (hcd->self.controller, "not queuing rh status urb\n");
611         retval = -EINVAL;
612     } else {
613         hcd->status_urb = urb;
614         urb->hcpriv = hcd;      /* indicate it's queued */
615
616         if (!hcd->uses_new_polling)
617             mod_timer (&hcd->rh_timer, jiffies +
618                         msecs_to_jiffies(250));
619
620         /* If a status change has already occurred, report it ASAP */
621         else if (hcd->poll_pending)
622             mod_timer (&hcd->rh_timer, jiffies);
623         retval = 0;
624     }
625     spin_unlock_irqrestore (&hcd_root_hub_lock, flags);
626     return retval;
627 }
```

还好这个函数不那么变态，由于设置了 `hcd->uses_new_polling` 为 1，而 `hcd->poll_pending` 只有在一个地方被改变，即 `usb_hcd_poll_rh_status()`，如果这个函数被调用了而 Hub 端口处没什么变化，那么 `poll_pending` 就会设置为 1。但当第一次来到这个函数时，`poll_pending` 还没有被设定过，则它只能是 0。

假设第一次执行 `usb_hcd_poll_rh_status` 时，Root Hub 的端口确实没有什么信息，即没有连接任何 USB 设备并且没有任何需要汇报的信息，那么 `poll_pending` 就会设置为 1。所以下一次当来到这个函数时，622 行的 `mod_timer` 会被执行。所以将再一次执行 `usb_hcd_poll_rh_status`，并且是立即执行！

但关于 `usb_hcd_poll_rh_status`，咱们也没什么好讲的，当初我们已经详细地讲过了。所以基本上我们就知道了，如果 Root Hub 的端口没有什么改变的话，`usb_submit_urb` 为 Root Hub 而提交的中断 `urb` 也不干什么正经事，我们能看到的是 `rh_queue_status`，`rh_urb_enqueue`，`usb_hcd_submit_urb`，`usb_submit_urb` 这四个函数像多米诺骨牌一样一个一个地依次返回 0。即使 Hub 端口里永远不接入任何设备，驱动程序也仍然大喊道“我们还活着”。

不过我最后想提醒一点，由于 Hub 驱动中的 `usb_submit_urb` 是在 `hub_probe` 的过程中被执行的，而这时候实际上正处在 `register_root_hub` 中，也就是说，咱们是在 `usb_add_hcd` 中，回过去看这个函数你会发现，1639 行调用 `register_root_hub`，而 1643 行调用 `usb_hcd_poll_rh_status`。这也就是说，尽管很早之前就讲过了 `usb_hcd_poll_rh_status` 函数，但是实际上第一次调用 `usb_hcd_poll_rh_status` 发生在 `rh_queue_status` 之后。这也就是为什么这里第一次进入 `rh_queue_status` 时，`poll_pending` 的值为 0，因为只有调用了 `usb_hcd_poll_rh_status` 之后，`poll_pending` 才有可能变成 1。

---

## 20. 非 Root Hub 的中断传输

再来看非 Root hub 的中断传输，`usb_submit_urb` 还是那个 `usb_submit_urb`，`usb_hcd_submit_urb` 还是那个 `usb_hcd_submit_urb`，但是很显然 `rh_urb_enqueue` 不会再被调用。取而代之的是 1014 行，`driver->urb_enqueue` 的被调用，即 `uhci_urb_enqueue` 函数。这个函数在讲控制传输时已经讲过了，后来在讲批量传输时又讲过，但是当时的上下文是控制传输或者批量传输，当然和现在的中断传输不一样。

我们回过头来看 `uhci_urb_enqueue`，很快就会发现对于中断传输，执行 1415 行，会调用 `uhci_submit_interrupt` 函数。于是有人建议我们立即去看 `uhci_submit_interrupt`，不过“有时候看到的不一定是真的，真的不一定看得到”。1415 行只是表面现象。

其实在看 `uhci_submit_interrupt` 之前，需要注意的是 1401 行，`uhci_alloc_qh` 函数。虽然大家都调用了它，可是不同的上下文里它做的事情大不一样。让我们再次回到 `uhci_alloc_qh` 中来，来自 `drivers/usb/host/uhci-q.c` 的函数不长，所以不妨再一次贴出来：

```
247 static struct uhci_qh *uhci_alloc_qh(struct uhci_hcd *uhci,  
248                                     struct usb_device *udev, struct usb_host_endpoint *hep)
```

```

249 {
250     dma_addr_t dma_handle;
251     struct uhci_qh *qh;
252
253     qh = dma_pool_alloc(uhci->qh_pool, GFP_ATOMIC, &dma_handle);
254     if (!qh)
255         return NULL;
256
257     memset(qh, 0, sizeof(*qh));
258     qh->dma_handle = dma_handle;
259
260     qh->element = UHCI_PTR_TERM;
261     qh->link = UHCI_PTR_TERM;
262
263     INIT_LIST_HEAD(&qh->queue);
264     INIT_LIST_HEAD(&qh->node);
265
266     if (udev) {                /* Normal QH */
267         qh->type = hep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK;
268         if (qh->type != USB_ENDPOINT_XFER_ISOC) {
269             qh->dummy_td = uhci_alloc_td(uhci);
270             if (!qh->dummy_td) {
271                 dma_pool_free(uhci->qh_pool, qh, dma_handle);
272                 return NULL;
273             }
274         }
275         qh->state = QH_STATE_IDLE;
276         qh->hep = hep;
277         qh->udev = udev;
278         hep->hcpriv = qh;
279
280         if (qh->type == USB_ENDPOINT_XFER_INT ||
281             qh->type == USB_ENDPOINT_XFER_ISOC)
282             qh->load = usb_calc_bus_time(udev->speed,
283                 usb_endpoint_dir_in(&hep->desc),
284                 qh->type == USB_ENDPOINT_XFER_ISOC,
285                 le16_to_cpu(hep->desc.wMaxPacketSize))
286                 / 1000 + 1;
287
288     } else {                    /* Skeleton QH */
289         qh->state = QH_STATE_ACTIVE;
290         qh->type = -1;
291     }
292     return qh;
293 }

```

很显然，280 行，不管是中断传输还是等时传输，都需要执行 282 行至 286 行这一小段。这一段其实就是调用了 `usb_calc_bus_time()` 函数。

这个函数来自 `drivers/usb/core/hcd.c`：

```

860 long usb_calc_bus_time (int speed, int is_input, int isoc, int Bytecount)
861 {
862     unsigned long    tmp;
863
864     switch (speed) {
865         case USB_SPEED_LOW:        /* INTR only */

```



```

866         if (is_input) {
867             tmp = (67667L * (31L + 10L * BitTime (Bytecount))) / 1000L;
868             return (64060L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
869         } else {
870             tmp = (66700L * (31L + 10L * BitTime (Bytecount))) / 1000L;
871             return (64107L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
872         }
873     case USB_SPEED_FULL:    /* ISOC or INTR */
874         if (isoc) {
875             tmp = (8354L * (31L + 10L * BitTime (Bytecount))) / 1000L;
876             return (((is_input) ? 7268L : 6265L) + BW_HOST_DELAY + tmp);
877         } else {
878             tmp = (8354L * (31L + 10L * BitTime (Bytecount))) / 1000L;
879             return (9107L + BW_HOST_DELAY + tmp);
880         }
881     case USB_SPEED_HIGH:    /* ISOC or INTR */
882         // FIXME adjust for input vs output
883         if (isoc)
884             tmp = HS_NSECS_ISO (Bytecount);
885         else
886             tmp = HS_NSECS (Bytecount);
887         return tmp;
888     default:
889         pr_debug ("%s: bogus device speed!\n", usbcore_name);
890         return -1;
891     }
892 }

```

这俨然就是一道小学数学题。传递进来的四个参数都很直白。`speed` 表征设备的速度，`is_input` 表征传输的方向，`isoc` 表征是不是等时传输，为 1 就是等时传输，为 0 则是中断传输。`Bytecount` 更加直白，要传输多少个 Bytes 的字节。

以前我一直只知道什么是贷款，因为我们 80 后中的大部分都不得不贷款去做房奴，但我从不知道究竟什么是带宽，看到了这个 `usb_calc_bus_time()` 函数和我们即将要看到的 `uhci_reserve_bandwidth()` 函数之后我总算是对带宽有一点了解了。

“带宽”这个词用“江湖”上的话来说，就是单位时间内传输的数据量，即单位时间内最大可能提供多少个二进制位传输，按“江湖”规矩，单位时间指的就是每秒。既然扯到时间，自然就应该计算时间。从软件的角度来说，每次建立一个管道我们都需要计算它所消耗的总线时间，或者说带宽，如果带宽不够了当然就不能建立了。

事实上以上这一堆的计算都是依据 USB spec 2.0 中 5.11.3 节里提供的公式，我们这里列举出全速的情况，spec 中的公式如图 3.16.1 所示：

## Full-speed (输入)

非等时传输 (包含 握手通信)

$$= 9107 + (83.54 + \text{Floor}(3.167 + \text{BitStuffTime}(\text{Data\_ba}))) + \text{Host\_Delay}$$

等时传输 (不包含握手通信)

$$= 7268 + (83.54 + \text{Floor}(3.167 + \text{BitStuffTime}(\text{Data\_ba}))) + \text{Host\_Delay}$$

## Full-speed (输出)

非等时传输 (包含 握手通信)

$$= 9107 + (83.54 + \text{Floor}(3.167 + \text{BitStuffTime}(\text{Data\_ba}))) + \text{Host\_Delay}$$

等时传输 (不包含握手通信)

$$= 6265 + (83.54 + \text{Floor}(3.167 + \text{BitStuffTime}(\text{Data\_ba}))) + \text{Host\_Delay}$$

图 3.16.1 全速设备的计算公式

这一切的单位都是 nm。

其中 BW\_HOST\_DELAY 是定义于 drivers/usb/core/hcd.h 的宏：

```
318 #define BW_HOST_DELAY 1000L /* nanoseconds */
```

它被定义为 1000L。BW 就是 BandWidth。这个宏对应于 spec 中的 Host\_Delay。而 BitTime 对应于 spec 中的 BitStuffTime，仔细对比这个函数和 spec 中的这一堆公式，你会发现，这个函数真是一点创意也没有，完全是按照 spec 来办事。所以写代码的这些人如果来参加大学生挑战杯，那么等待他们的只能是早早被淘汰，连上 PK 台的机会都甭想有。

总之，这个函数返回的就是传输这些字节将会占有多少时间，单位是纳秒。在我们的故事中，usb\_calc\_bus\_time 这个计算时间的函数只会出现在这一个地方，即每次咱们调用 uhci\_alloc\_qh 申请一个正常的 QH 时会被调用。（最开始建立框架时当然不会被调用。）而且只有针对中断传输和等时传输才需要申请带宽。这里我们把返回值赋给了 qh->load，赋值之前我们除了 1000，即把单位转换成为了微秒。

于是又结束 uhci\_alloc\_qh，回到了 uhci\_urb\_enqueue。当然我还想友情提醒一下，uhci\_alloc\_qh 中，第 267 行，对 qh->type 赋了值，这个值来自 struct usb\_host\_endpoint 结构体指针 hep，确切的说就是来自于端点描述符中的 bmAttributes 这一项。USB spec 2.0 中规定好了，这个属性的 Bit1 和 Bit0 两位表征了端点的传输类型。00 为控制，01 为等时，10 为 Bulk，11 为 Interrupt，而咱们这里的 USB\_ENDPOINT\_XFERTYPE\_MASK 就是为了提取出这两个 bit 来。

于是回到 uhci\_urb\_enqueue 中以后，对 qh->type 进行判断，如果是中断传输类型，则 uhci\_submit\_interrupt 会被调用，依然来自 drivers/usb/host/uhci-q.c：

```

1060 static int uhci_submit_interrupt(struct uhci_hcd *uhci, struct urb *urb,
1061                                struct uhci_qh *qh)
1062 {
1063     int ret;
1064
1065     /* USB 1.1 interrupt transfers only involve one packet per interval.
1066      * Drivers can submit URBs of any length, but longer ones will need
1067      * multiple intervals to complete.
1068      */
1069
1070     if (!qh->bandwidth_reserved) {
1071         int exponent;
1072
1073         /* Figure out which power-of-two queue to use */
1074         for (exponent = 7; exponent >= 0; --exponent) {
1075             if ((1 << exponent) <= urb->interval)
1076                 break;
1077         }
1078         if (exponent < 0)
1079             return -EINVAL;
1080         qh->period = 1 << exponent;
1081         qh->skel = SKEL_INDEX(exponent);
1082
1083         /* For now, interrupt phase is fixed by the layout
1084          * of the QH lists. */
1085         qh->phase = (qh->period / 2) & (MAX_PHASE - 1);
1086         ret = uhci_check_bandwidth(uhci, qh);
1087         if (ret)
1088             return ret;
1089     } else if (qh->period > urb->interval)
1090         return -EINVAL; /* Can't decrease the period */
1091
1092     ret = uhci_submit_common(uhci, urb, qh);
1093     if (ret == 0) {
1094         urb->interval = qh->period;
1095         if (!qh->bandwidth_reserved)
1096             uhci_reserve_bandwidth(uhci, qh);
1097     }
1098     return ret;
1099 }

```

首先 `struct uhci_qh` 中有一个成员 `unsigned int bandwidth_reserved`，顾名思义，用来表征是否申请了带宽，对于等时传输和中断传输，需要为之分配带宽，带宽就是占用总线的时间，UHCI 的世界里，等时传输和中断传输这两者在每一个 Frame 内加起来是不可以超过该 Frame 的 90% 的。

设置 `bandwidth_reserved` 为 1 只有一个地方，那就是 `uhci_reserve_bandwidth` 函数。而与之相反的一个函数 `uhci_release_bandwidth` 会把这个变量设置为 0。而调用 `uhci_reserve_bandwidth` 的又是谁呢？只有两个地方：一个恰恰就是这里的 `uhci_submit_interrupt`，另一个则是等时传输中要用到的 `uhci_submit_isochronous`。而释放这个带宽的函数 `uhci_release_bandwidth` 则在 `uhci_giveback_urb` 中被调用。

1074 行，临时变量 `exponent` 从 7 开始，最多循环 8 次，把 1 左移 `exponent` 位就是进行指

数运算，比如 exponent 为 1，左移以后就是 2 的 1 次方，exponent 为 7，则左移以后就是 2 的 7 次方。把这个数和 urb->interval 相比较，如果小于等于 urb->interval，就算找到了。这是什么意思呢？我们知道，UHCI 是 USB spec 1.1 的产物，那时候只有全速和低速设备，而 USB spec 中规定，对于全速设备来说，其 interval 必须在 1 毫秒到 255 毫秒之间，对于低速设备来说，其 interval 必须在 10 毫秒到 255 毫秒之间，所以这里 exponent 最多取 7 即可，2 的 7 次方就是 128。如果 interval 比 128 还大，那么就是处于 128 至 255 之间。而 interval 最小也不能小于 1，小于 1 也就出错了。那么从 1074 行到 1080 行这一段的目的是什么呢？就是根据 interval 确定最终的周期，就是说甭管您 interval 具体是多少，最终设定的周期都是 2 的整数次方，只要周期小于等于 interval，设备驱动就不会有意见。

SKEL\_INDEX 这个宏我们贴出来过，struct uhci\_qh 有一个成员 int skel，qh->skel 将被赋值为 9-exponent，即比如 exponent 为 1，qh->skel 就是 8。但同时我们知道，比如 exponent 为 3，那么说明 urb->interval 是介于 8 毫秒和 16 毫秒之间。而 qh->skel 为 8 意味着 qh 最终将挂在 skelqh[] 数组的 skel int8 QH 后面。

此外，struct uhci\_qh 另有两个元素：unsigned int period 和 short phase，刚才说了 period 就是周期，这里看到它被赋值为 2 的 exponent 次方，即比如 exponent 为 3，那么 period 就是 8。我们知道，标准情况下一个 Frame 是 1 毫秒，所以对于中断传输来说，这里的意思就是每 8 个 Frame 主机关心一次设备。MAX\_PHASE 被定义为 32，此时我们还看不出来 phase 这个变量有什么用，到时候再看。

前面我们计算的是总线时间，现在还得转换成带宽的概念。uhci\_check\_bandwidth 函数就是用来检查带宽的，它来自 drivers/usb/host/uhci-q.c:

```

627 static int uhci_check_bandwidth(struct uhci_hcd *uhci, struct uhci_qh *qh)
628 {
629     int minmax_load;
630
631     /* Find the optimal phase (unless it is already set) and get
632      * its load value. */
633     if (qh->phase >= 0)
634         minmax_load = uhci_highest_load(uhci, qh->phase, qh->period);
635     else {
636         int phase, load;
637         int max_phase = min_t(int, MAX_PHASE, qh->period);
638
639         qh->phase = 0;
640         minmax_load = uhci_highest_load(uhci, qh->phase, qh->period);
641         for (phase = 1; phase < max_phase; ++phase) {
642             load = uhci_highest_load(uhci, phase, qh->period);
643             if (load < minmax_load) {
644                 minmax_load = load;
645                 qh->phase = phase;
646             }
647         }
648     }
649 }
```

```

650      /* Maximum allowable periodic bandwidth is 90%, or 900 us per frame*/
651      if (minimax_load + qh->load > 900) {
652          dev_dbg(uhci_dev(uhci), "bandwidth allocation failed: "
653                  "period %d, phase %d, %d + %d us\n",
654                  qh->period, qh->phase, minimax_load, qh->load);
655          return -ENOSPC;
656      }
657      return 0;
658 }

```

在提交中断类型的 urb 或者是等时类型的 urb 时，需要检查带宽，看带宽够不够了。这种情况下这个函数就会被调用。这个函数正常的话就将返回 0，负责就返回错误码-ENOSPC。不过你别小看这个函数，做程序员的最高境界就是像和尚研究佛法一样研究算法！所以写代码的人在这里用代码体现了他的境界。我们来仔细分析一下这个函数。

633 行判断 qh->phase 是否小于零，在 uhci\_submit\_interrupt 中设置了 qh->phase，显然从这个上下文来看 qh->phase 一定是大于等于 0 的，不过您别忘了，检测带宽这件事情在提交等时类型的 urb 时也会被调用，到时候你会发现，我们会把 qh->phase 设置为-1。所以再回过头来看这个函数，而现在，635 到 648 这一段先略过，因为现在不会被执行。现在只要关注 634 行就够了。uhci\_highest\_load 这个函数来自 drivers/usb/host/uhci-q.c:

```

614 static int uhci_highest_load(struct uhci_hcd *uhci, int phase, int period)
615 {
616     int highest_load = uhci->load[phase];
617
618     for (phase += period; phase < MAX_PHASE; phase += period)
619         highest_load = max_t(int, highest_load, uhci->load[phase]);
620     return highest_load;
621 }

```

代码本身超级简单，难的是这代码背后的哲学。struct uhci\_hcd 有一个成员，short load[MAX\_PHASE]，前面说过，MAX\_PHASE 就是 32。所以这里就是为每一个 UHCI 主机控制器准备这么一个数组，来记录它的负载。这个数组 32 个元素，每一个元素就代表一个 Frame，所以这个数组实际上就是记录了一个主机控制器的 32 个 Frame 内的负载。我们知道一个 UHCI 主机控制器对应 1024 个 Frame 组成的 Frame List。但是软件角度来说，本着建设节约型社会的原则，没有必要申请一个 1024 的元素的数组，所以就申请 32 个元素。这个数组被称为“periodic load table”。于是这个函数所做的就是以 period 为步长，找到这个数组中最大的元素，即该 Frame 的负载最重。

得到这个最大的负载所对应的 Frame 之后，在 651 行计算这个负载加上刚才计算总线时间得到的那个 qh->load，这两个值不能超过 900，单位是微秒，因为一个 Frame 是一个毫秒，而 USB spec 规定了，等时传输和中断传输所占的带宽不能超过一个 Frame 的 90%。道理很简单，资源都被它们俩占了，别人就没法混了。无论如何也要为批量传输和控制传输着想一下。

于是 uhci\_check\_bandwidth 结束了，这里会调用 uhci\_submit\_common，这个函数在批量传输中已经讲过了，这是它们之间的公共函数，其执行过程也和批量传输一样，无非是通过 urb

得到 TD，依次调用 `uhci_alloc_td`，`uhci_add_td_to_urbp` 和 `uhci_fill_td`，完了之后设置最后一个 TD 的中断标志 `TD_CTRL_IOC`。

然后，`uhci_submit_common` 结束之后回到 `uhci_submit_interrupt`，剩下的代码也不多了，正常时返回 0，于是设置 `urb->interval` 为 `qh->period`，没有保留带宽就执行 `uhci_reserve_bandwidth` 去保留带宽。仍然来自 `drivers/usb/host/uhci-q.c`：

```

663 static void uhci_reserve_bandwidth(struct uhci_hcd *uhci, struct uhci_qh
*qh)
664 {
665     int i;
666     int load = qh->load;
667     char *p = "??";
668
669     for (i = qh->phase; i < MAX_PHASE; i += qh->period) {
670         uhci->load[i] += load;
671         uhci->total_load += load;
672     }
673     uhci_to_hcd(uhci)->self.bandwidth_allocated =
674         uhci->total_load / MAX_PHASE;
675     switch (qh->type) {
676     case USB_ENDPOINT_XFER_INT:
677         ++uhci_to_hcd(uhci)->self.bandwidth_int_reqs;
678         p = "INT";
679         break;
680     case USB_ENDPOINT_XFER_ISOC:
681         ++uhci_to_hcd(uhci)->self.bandwidth_isoc_reqs;
682         p = "ISO";
683         break;
684     }
685     qh->bandwidth_reserved = 1;
686     dev_dbg(uhci_dev(uhci),
687             "%s dev %d ep%02x-%s, period %d, phase %d, %d us\n",
688             "reserve", qh->udev->devnum,
689             qh->hep->desc.bEndpointAddress, p,
690             qh->period, qh->phase, load);
691 }

```

其实这个函数也挺简单的。`uhci->load` 数组就是在这个函数这里被赋值的。当然它的“情侣”函数 `uhci_release_bandwidth` 里面也会改变这个数组。而 `uhci->total_load` 则是把所有的负担（load）全都加到一起。而 `bandwidth_allocated` 则是 `total_load` 除以 32，即一个平均值。

然后根据 QH 是中断类型还是等时类型，分别增加 `bandwidth_int_reqs` 和 `bandwidth_isoc_reqs`。这两个都是 `struct usb_bus` 的 `int` 类型成员，前者记录中断请求的数量，后者记录等时请求的数量。

最后设置 `qh->bandwidth_reserved` 为 1。这个函数就结束了。这样，`uhci_submit_interrupt` 这个函数也结束了。终于回到了 `uhci_urb_enqueue`。

1426 行，把 `qh` 赋给 `urbp` 的 `qh`。

把 `urbp` 给链入到 `qh` 的队列中来。`qh` 里面专门有一个队列记录它所领导的各个 `urbp`。因为一个端点对应一个 `QH`，而该端点可以有多个 `urb`，所以就把它们都排成一个队列。

1433 行，如果这个队列的下一个节点就是现在这个 `urbp`，并且 `qh` 没有停止，则调用 `uhci_activate_qh()` 和 `uhci_urbp_wants_fsbr()`。这两个函数当初在控制传输中就已经讲过了，不过对于 `uhci_activate_qh()` 现在进去看会有所不同。

在 514 行开始的这一小段判断中，看到的是对 `qh->skel` 进行的判断，这是一个 `int` 型的变量，当初在 `uhci_submit_interrupt` 中对这个变量进行了赋值，赋的值是 `SKEL_INDEX(exponent)`。很显然它小于 `SKEL_ASYNC`，所以这里 `link_interrupt` 会被执行。这个函数来自 `drivers/usb/host/uhci-q.c`：

```
439 static void link_interrupt(struct uhci_hcd *uhci, struct uhci_qh *qh)
440 {
441     struct uhci_qh *pqh;
442
443     list_add_tail(&qh->node, &uhci->skelqh[qh->skel]->node);
444
445     pqh = list_entry(qh->node.prev, struct uhci_qh, node);
446     qh->link = pqh->link;
447     wmb();
448     pqh->link = LINK_TO_QH(qh);
449 }
```

把 `qh` 的 `node` 给链入到 `uhci->skelqh[qh->skel]` 的 `node` 链表中去。然后让这个 `qh` 的 `link` 指向前一个 `qh` 的 `link`，并且把前一个 `qh` 的 `link` 指针指向这个 `qh`。这就是典型的队列插入的操作。很明显这里又是物理地址的链接。

`uhci_activate_qh` 就算执行完了。剩下的代码就和控制传输/批量传输一样了。`uhci_urb_enqueue` 也就这样结束了，`usb_hcd_submit_urb` 啊，`usb_submit_urb` 啊，也纷纷跟着结束了。似乎调用 `usb_submit_urb` 提交了一个中断请求的 `urb` 之后整个世界没有发生任何变化，完全没有看出咱们这个函数对这个世界的影晌，俨然这个函数的调用没有任何意义，但我要告诉你，其实不是的，这次函数调用就像流星，短暂地划过却能照亮整个天空。此刻，让我们利用 `debugfs` 来看个究竟，当我们没有提交任何 `urb` 时，`/sys/kernel/debug/uhci` 目录下面的文件是这个样子的：

```
localhost:~ # cat /sys/kernel/debug/uhci/0000\:\00\:\1d.1
Root-hub state: suspended  FSBR: 0
HC status
usbcmd    =    0048  Maxp32 CF EGSM
usbstat    =    0020  HCHalted
usbint     =    0002
usbfrnum   = (1)168
flbaseadd = 194a9168
sof        =     40
stat1      =    0080
stat2      =    0080
Most recent frame: 45a (90)  Last ISO frame: 45a (90)
```

```

Periodic load table
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
Total: 0, #INT: 0, #ISO: 0
Frame List
Skeleton QHs
- skel_unlink_qh
  [d91a1000] Skel QH link (00000001) element (00000001)
  queue is empty
- skel_iso_qh
  [d91a1060] Skel QH link (00000001) element (00000001)
  queue is empty
- skel_int128_qh
  [d91a10c0] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int64_qh
  [d91a1120] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int32_qh
  [d91a1180] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int16_qh
  [d91a11e0] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int8_qh
  [d91a1240] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int4_qh
  [d91a12a0] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int2_qh
  [d91a1300] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_async_qh
  [d91a1360] Skel QH link (00000001) element (197bd000)
  queue is empty
[d97bd000] link (00000001) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=7f, PID=69 (IN)
(buf=00000000)
- skel_term_qh
  [d91a13c0] Skel QH link (191a13c2) element (197bd000)
  queue is empty

```

可以看到，那 11 个 **skel QH** 都被打印了出来，**link** 后面的括号里面的东西是 **link** 的地址，**element** 后面的括号里面的东西是 **element** 的地址。这个时候整个调度框架中没有任何有实质意义的 **QH** 或者 **TD**。

**Periodic load table** 后面打印出来的是 **uhci->load[]** 数组的 32 个元素，看到这时候这 32 个元素全是 0，因为目前没有任何中断调度或者等时调度。下面我们做一个实验，往 USB 端口里插入一个 USB 键盘，然后加载其驱动程序，比如 **usbhid** 模块，再来看同一个文件：

```

localhost:~ # cat /sys/kernel/debug/uhci/0000\:00\:1d.1
Root-hub state: running  FSRB: 0
HC status

```



```

usbcmd = 00c1 Maxp64 CF RS
usbstat = 0000
usbint = 000f
usbfrnum = (1)a70
flbaseadd = 194a9a70
sof = 40
stat1 = 0080
stat2 = 01a5 LowSpeed Enabled Connected
Most recent frame: 8ae66 (614) Last ISO frame: 8ae66 (614)
Periodic load table
    0      0      0      0      0      0      0      0
   118    0      0      0      0      0      0      0
    0      0      0      0      0      0      0      0
   118    0      0      0      0      0      0      0
Total: 236, #INT: 1, #ISO: 0
Frame List
Skeleton QHs
- skel_unlink_qh
  [d91a1000] Skel QH link (00000001) element (00000001)
  queue is empty
- skel_iso_qh
  [d91a1060] Skel QH link (00000001) element (00000001)
  queue is empty
- skel_int128_qh
  [d91a10c0] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int64_qh
  [d91a1120] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int32_qh
  [d91a1180] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int16_qh
  [d91a11e0] Skel QH link (191a1482) element (00000001)
  queue is empty
  [d91a1480] INT QH link (191a1362) element (197bd0c0)
  period 16 phase 8 load 118 us
urb_priv [d4b31720] urb [d9d84440] qh [d91a1480] Dev=2 EP=1(IN) INT Actlen=0
  1: [d97bd0c0] link (197bd030) e3 LS IOC Active NAK Length=7ff MaxLen=7
DT0 EndPt=1 Dev=2, PID=69(IN) (buf=18c69000)
  Dummy TD
  [d97bd030] link (197bd060) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=0, PID=e1(OUT)
(buf=00000000)
- skel_int8_qh
  [d91a1240] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int4_qh
  [d91a12a0] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_int2_qh
  [d91a1300] Skel QH link (191a1362) element (00000001)
  queue is empty
- skel_async_qh
  [d91a1360] Skel QH link (00000001) element (197bd000)
  queue is empty
  [d97bd000] link (00000001) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=7f, PID=69(IN)
(buf=00000000)
- skel_term_qh
  [d91a13c0] Skel QH link (191a13c2) element (197bd000)

```

```
queue is empty
```

最显著的两个变化是：第一，Periodic load table 这张表不再全是 0 了；第二，在 skel\_int16\_qh 下面不再是空空如也了，有一个 int QH 了，有一个 urb\_priv 了，这个 int QH 的周期是 16，phase 是 8，load 是 118 $\mu$ s。对照 Periodic load table，再结合这三个数字，你是不是能明白 phase 的含义了。没错，load 这个数组一共 32 个元素，编号从 0 开始到 31 结束，周期是 16 就意味着每隔 16 ms 这个中断传输会被调度一次，phase 是 8 就意味着它的起始点位于编号为 8 的位置，即从 8 开始，8，24，40，56，……每隔 16 ms 就安置一个中断传输的调度。而 118 $\mu$ s 则是它在每个 Frame 中占据总线的时间。

至此，既了解了中断传输的处理，也了解了 debugfs 在 uhci-hcd 模块中的应用。文件 drivers/usb/host/uhci-debug.c 一共 592 行就是努力让我们能够在/sys/kernel/debug/目录下面看到刚才这些信息。实际上通过以上 sysfs 提供的信息，对于整个 uhci-hcd 的结构也有了很好的了解，之前的任何一个数据结构，比如 skel\_term\_qh，Dummy\_TD，整个 skelqh 数组，link，element，periodic load table 这一切的一切，都通过这些信息展现得淋漓尽致。也正是通过这些信息，我们才真正体会到了 skelqh 这个数组的意义和价值，没有它们构建的基础框架，毫无疑问，在 uhci-hcd 中使用 skelqh 这个数组是一个无比英明的决定。尽管有人觉得 skelqh 的存在浪费了内存，而且搞得代码看上去复杂了许多，但它确实非常实用。

## 21. 等时传输

由于等时传输的特殊性，很多地方它都被特别地对待了。从 usb\_submit\_urb 开始就显示出了它的与众不同了。该函数中 268 行，判断 temp 是不是 PIPE\_ISOCHRONOUS，即是不是等时传输，如果是，就执行下面那段代码。

278 行，int number\_of\_packets 是 struct urb 的一个成员，它用来指定该 urb 所处理的等时传输缓冲区的数量，或者说这个等时传输要传输多少个 packet，每一个 packet 用一个 struct usb\_iso\_packet\_descriptor 结构体变量来描述。对于每一个 packet，需要建立一个 td。

同时，还注意到 struct urb 有另外一个成员，struct usb\_iso\_packet\_descriptor iso\_frame\_desc[0]，又是一个零长度数组，这个数组用来帮助这个 urb 定义多个等时传输，而这个数组的实际长度恰恰就是我们前面提到的那个 number\_of\_packets。设备驱动程序在提交等时传输 urb 时，必须设置好 urb 的 iso\_frame\_desc 数组。有人提问，为何 iso\_frame\_desc 数组的长度恰好是 number\_of\_packets？从哪里看出来的？还记得很久很久以前，曾经讲过一个叫做 usb\_alloc\_urb() 的函数么？不管是在 usb-storage 中还是在 Hub 驱动中，都见过这个函数，它的作用就是申请 urb，或许忘记了这个函数的参数，在 include/linux/usb.h 中可找到它的原型：

```
1266 extern struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
```

这其中第一个参数，iso\_packets，其实就是咱们这里的 number\_of\_packets。所以，设备驱动在申请等时 urb 时，必须指定需要传输多少个 packets。usb\_alloc\_urb()，来自 drivers/usb/core/urb.c:

```
56 struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
57 {
58     struct urb *urb;
59
60     urb = kmalloc(sizeof(struct urb) +
61                 iso_packets * sizeof(struct usb_iso_packet_descriptor),
62                 mem_flags);
63     if (!urb) {
64         err("alloc_urb: kmalloc failed");
65         return NULL;
66     }
67     usb_init_urb(urb);
68     return urb;
69 }
```

再一次看到了零长度数组的应用，或者叫做变长度数组的应用。struct usb\_iso\_packet\_descriptor 的定义来自 include/linux/usb.h:

```
952 struct usb_iso_packet_descriptor {
953     unsigned int offset;
954     unsigned int length;          /* expected length */
955     unsigned int actual_length;
956     int status;
957 };
```

这个结构体的意思很简洁明了。这个结构体描述的就是一个 iso 包。而 urb 的 iso\_frame\_desc 数组的元素都在设备驱动提交 urb 之前就设置好了。其中 length 就如注释里说的一样，是期待长度。而 actual\_length 是实际长度，这里我们先把它设置为 0。

至于 348 行，对于高速设备，如果 urb->interval 大于 1024×8，则设置为 1024×8，注意这里单位是微帧，即 125μs，以及 360 行，对于全速设备的 ISO 传输，如果 urb->interval 大于 1024，则设置为 1024，注意这里单位是帧，即 1 ms。关于这两条，Alan Stern 的解释是，由于主机控制器驱动中并不支持超过 1024 个 ms 的间隔，（想想也很简单，比如 UHCI 吧，总共 Frame List 才 1024 个元素，你这个间隔期总不能超过它吧，要不还不乱了去）。

进入 usb\_hcd\_submit\_urb，因为 Root Hub 是不会有等时传输的，所以针对非 Root Hub，调用 uhci\_urb\_enqueue。1419 行，调用 uhci\_submit\_isochronous()。这个函数来自 drivers/usb/host/uhci-q.c:

```
1231 static int uhci_submit_isochronous(struct uhci_hcd *uhci, struct urb *urb,
1232                                   struct uhci_qh *qh)
1233 {
1234     struct uhci_td *td = NULL;    /* Since urb->number_of_packets > 0 */
1235     int i, frame;
```

```

1236     unsigned long destination, status;
1237     struct urb_priv *urbp = (struct urb_priv *) urb->hcpriv;
1238
1239     /* Values must not be too big (could overflow below) */
1240     if (urb->interval >= UHCI_NUMFRAMES ||
1241         urb->number_of_packets >= UHCI_NUMFRAMES)
1242         return -EFBIG;
1243
1244     /* Check the period and figure out the starting frame number */
1245     if (!qh->bandwidth_reserved) {
1246         qh->period = urb->interval;
1247         if (urb->transfer_flags & URB_ISO_ASAP) {
1248             qh->phase = -1; /* Find the best phase */
1249             i = uhci_check_bandwidth(uhci, qh);
1250             if (i)
1251                 return i;
1252
1253             /* Allow a little time to allocate the TDs */
1254             uhci_get_current_frame_number(uhci);
1255             frame = uhci->frame_number + 10;
1256
1257             /* Move forward to the first frame having the
1258              * correct phase */
1259             urb->start_frame = frame + ((qh->phase - frame) &
1260                                         (qh->period - 1));
1261         } else {
1262             i = urb->start_frame - uhci->last_iso_frame;
1263             if (i <= 0 || i >= UHCI_NUMFRAMES)
1264                 return -EINVAL;
1265             qh->phase = urb->start_frame & (qh->period - 1);
1266             i = uhci_check_bandwidth(uhci, qh);
1267             if (i)
1268                 return i;
1269         }
1270
1271     } else if (qh->period != urb->interval) {
1272         return -EINVAL; /* Can't change the period */
1273
1274     } else { /* Pick up where the last URB leaves off */
1275         if (list_empty(&qh->queue)) {
1276             frame = qh->iso_frame;
1277         } else {
1278             struct urb *lurb;
1279
1280             lurb = list_entry(qh->queue.prev,
1281                               struct urb_priv, node)->urb;
1282             frame = lurb->start_frame +
1283                     lurb->number_of_packets *
1284                     lurb->interval;
1285         }
1286         if (urb->transfer_flags & URB_ISO_ASAP)
1287             urb->start_frame = frame;
1288         else if (urb->start_frame != frame)
1289             return -EINVAL;
1290     }
1291
1292     /* Make sure we won't have to go too far into the future */
1293     if (uhci_frame_before_eq(uhci->last_iso_frame + UHCI_NUMFRAMES,
1294                             urb->start_frame + urb->number_of_packets *

```

```

1295         urb->interval))
1296     return -EFBIG;
1297
1298     status = TD_CTRL_ACTIVE | TD_CTRL_IOS;
1299     destination=(urb->pipe & PIPE_DEVEP_MASK)|usb_packetid(urb->pipe);
1300
1301     for (i = 0; i < urb->number_of_packets; i++) {
1302         td = uhci_alloc_td(uhci);
1303         if (!td)
1304             return -ENOMEM;
1305
1306         uhci_add_td_to_urbp(td, urbp);
1307         uhci_fill_td(td, status, destination |
1308                     uhci_explen(urb->iso_frame_desc[i].length),
1309                     urb->transfer_dma +
1310                     urb->iso_frame_desc[i].offset);
1311     }
1312
1313     /* Set the interrupt-on-completion flag on the last packet. */
1314     td->status |= __constant_cpu_to_le32(TD_CTRL_IOC);
1315
1316     /* Add the TDs to the frame list */
1317     frame = urb->start_frame;
1318     list_for_each_entry(td, &urbp->td_list, list) {
1319         uhci_insert_td_in_frame_list(uhci, td, frame);
1320         frame += qh->period;
1321     }
1322
1323     if (list_empty(&qh->queue)) {
1324         qh->iso_packet_desc = &urb->iso_frame_desc[0];
1325         qh->iso_frame = urb->start_frame;
1326         qh->iso_status = 0;
1327     }
1328
1329     qh->skel = SKEL_ISO;
1330     if (!qh->bandwidth_reserved)
1331         uhci_reserve_bandwidth(uhci, qh);
1332     return 0;
1333 }

```

1240 行，UHCI\_NUMFRAMES 是 1024，同样，urb 的 interval 显然不能比这个还大，它的 number\_of\_packets 也不能比这个大，要不然肯定就溢出了。就像伤痛，当眼泪掉下来，一定是伤痛已经超载。

接下来看，URB\_ISO\_ASAP 这个 flag 是专门给等时传输用的，它的意思就是告诉驱动程序，只要带宽允许，那么就从此点开始设置这个 urb 的 start\_frame 变量。通常为了尽可能快地得到图像数据，应当在 urb 中指定这个 flag，因为它意味着尽可能快发出本 urb。比如说，之前有一个 urb，是针对 iso 端点的，假设它有两个 packets，被安排在 Frame 号 108 和 109，即假设其 interval 是 1。现在再假设新的一个 urb 是在 Frame 111 被提交的，如果设置了 URB\_ISO\_ASAP 这个 flag，那么这个 urb 的第一个 packet 就会在下一个可以接受的 Frame 中被执行，比如 Frame 112。但是如果没有设置这个 URB\_ISO\_ASAP 的 flag 呢？这个 packet 就会被安排在上一个 urb 结束之后的下一个 Frame，即 110。尽管 Frame 110 已经过去了，但是这种调度仍然有意义，因

为它可以保证一定接下来的 packets 处于特定的阶段, 因为有时, 驱动程序并不在乎丢掉一些包, 尤其是等时传输。

我们看到这里 qh 的 phase 被设置为-1。所以在 uhci\_check\_bandwidth 函数中有一个判断条件是 qh 的 phase 是否大于等于 0。如果调用 uhci\_check\_bandwidth 之前设置了 phase 大于等于 0, 则表明咱们手工设置了 phase, 否则可通过一种算法来选择出一个合适的 phase。这个函数正常应该返回 0。

接下来是 uhci\_get\_current\_frame\_number():

```
441 static void uhci_get_current_frame_number(struct uhci_hcd *uhci)
442 {
443     if (!uhci->is_stopped) {
444         unsigned delta;
445
446         delta = (inw(uhci->io_addr + USBFRNUM) - uhci->frame_number) &
447                (UHCI_NUMFRAMES - 1);
448         uhci->frame_number += delta;
449     }
450 }
```

UHCI 主机控制器有一个 Frame 计数器, Frame 从 0 到 1023, 然后又从 0 开始, 那么这个数到底是多少呢? 这个函数就是用来获得这个值的, 读了端口和 USBFRNUM 寄存器。uhci->frame\_number 用来记录 Frame Number, 所以这里的做法就是把当前的 Frame Number 减去上次保存在 uhci->frame\_number 中的值, 然后转换成二进制, 得到一个差值, 再更新 UHCI 的 frame\_number。

而 start\_frame 就是这个传输开始的 Frame。这里让 Frame 等于当前的 Frame 加上 10, 就是给个延时, 如注释所说的那样, 给内存申请一点点时间。再让 start\_frame 等于 Frame 加上 (qh->phase-frame) 和 (qh->period-1) 相与。熟悉二进制运算的同志们应该不难知道这样做最终得到的 start\_frame 是什么, 很显然, 它会满足 phase 的要求。

1261 行, else, 就是驱动程序指定了 start\_frame, 这种情况下就是直接设置 phase, last\_iso\_frame 就对应于刚才这个例子中的 frame 109。

1293 行, uhci\_frame\_before\_eq 就是一个普通的宏, 来自 drivers/usb/host/uhci-hcd.h:

```
441 /* Utility macro for comparing frame numbers */
442 #define uhci_frame_before_eq(f1, f2)    (0 <= (int) ((f2) - (f1)))
```

其实就是比较两个 Frame Number, 如果第 2 个比第 1 个大的话, 就返回真, 反之就返回假。而咱们这里代码的意思是, 如果第 2 个比第 1 个大, 那么说明出错了。last\_iso\_frame 是记录着上一次扫描时的 Frame 号, 在 uhci\_scan\_schedule 中会设置, UHCI\_NUMFRAMES 我们知道是 1024。urb 的 number\_of\_packets 与 interval 的乘积就表明将要花掉多少时间, 它们加上 urb 的 start\_frame 就等于这些包传输完之后的时间, 或者说 Frame Number。这里的意思就是希望一次

传输的东西别太大了，不能越界。-EFBIG 这个错误码的含义本身就是文件太大 (File too large)。

1298 行，TD\_CTRL\_IOS，对应于 TD 的 bit25，IOS 的意思是 Isochronous Select，这一位为 1 表示 TD 是一个等时传输描述符，即 Isochronous Transfer Descriptor。如果为 0 则表示这是一个非等时传输描述符。等时传输的 TD 在执行完之后会被主机控制器设置为 inactive，不管执行的结果是什么。下面还设置了 TD\_CTRL\_IOC，告诉主机控制器在 TD 执行的 Frame 结束时发送一个中断。

然后根据 packets 的数量申请 TD，再把本 urb 的各个 TD 加入到 frame list 中去。uhci\_insert\_td\_in\_frame\_list 是来自 drivers/usb/host/uhci-q.c:

```

159 static inline void uhci_insert_td_in_frame_list(struct uhci_hcd *uhci,
160          struct uhci_td *td, unsigned framenum)
161 {
162     framenum &= (UHCI_NUMFRAMES - 1);
163
164     td->frame = framenum;
165
166     /* Is there a TD already mapped there? */
167     if (uhci->frame_cpu[framenum]) {
168         struct uhci_td *ftd, *ltd;
169
170         ftd = uhci->frame_cpu[framenum];
171         ltd = list_entry(ftd->fl_list.prev, struct uhci_td, fl_list);
172
173         list_add_tail(&td->fl_list, &ftd->fl_list);
174
175         td->link = ltd->link;
176         wmb();
177         ltd->link = LINK_TO_TD(td);
178     } else {
179         td->link = uhci->frame[framenum];
180         wmb();
181         uhci->frame[framenum] = LINK_TO_TD(td);
182         uhci->frame_cpu[framenum] = td;
183     }
184 }
```

只有等时传输才需要使用这个函数。先看 else 这一段，让 td 在物理上指向 uhci 的 frame 数组中对应元素，framenum 是传递进来的参数，其实就是 urb 的 start\_frame。而 frame 数组里又设置为 td 的物理地址。要知道之前曾经在 configure\_hc 中把 frame 和实际硬件的 frame list 联系了起来，因此只要把 td 和 frame 联系起来就等于和硬件联系了起来，另一方面这里又把 frame\_cpu 和 td 联系起来，所以以后只要直接通过 frame\_cpu 来操作队列即可。正如下面在 if 段所看到的那样。

来看 if 这一段，struct uhci\_td 有一个成员 struct list\_head fl\_list，struct uhci\_hcd 中有一个成员 void \*\*frame\_cpu，当初在 uhci\_start 函数中为 uhci->frame\_cpu 申请好了内存，而刚才在 else 里面看到每次会把 frame\_cpu 数组的元素赋值为 td，所以这里就是把 td 通过 fl\_list 链入到 ftd 的 fl\_list 队列里去。而物理上，也把 td 给插入到这个队列中来。

如果 `qh` 的 `queue` 为空，即没有任何 `urb`，就设置 `qh` 的几个成员，`iso_packet_desc` 是下一个 `urb` 的 `iso_frame_desc`，`iso_frame` 则是该 `iso_packet_desc` 的 `frame` 号，`iso_status` 则是该 `iso urb` 的状态。

最后，令 `qh->skel` 等于 `SKEL_ISO`，然后调用 `uhci_reserve_bandwidth` 保留带宽。

至此，`uhci_submit_isochronous` 就结束了。回到 `uhci_urb_enqueue`，下一步执行，`uhci_activate_qh`，而在这个函数中，我们将调用 `link_iso`。

而 `link_iso` 同样来自 `drivers/usb/host/uhci-q.c`:

```
428 static inline void link_iso(struct uhci_hcd *uhci, struct uhci_qh *qh)
429 {
430     list_add_tail(&qh->node, &uhci->skel_iso_qh->node);
431
432     /* Isochronous QHs aren't linked by the hardware */
433 }
```

这就简单多了，直接加入到 `skel_iso_qh` 中去就可以了。

终于，四大传输也就这样结束了。而我们的故事也即将 ALT+F4 了。我只是说也许。

如果失败的人生可以 F5，如果莫名的悲伤可以 DEL；

如果逝去的岁月可以 CTRL+C，如果甜蜜的往事可以 CTRL+V；

如果一切都可以 CTRL+ALT+DEL，那么我们所有的故事是不是永远都不会 ALT+F4？

## 22. “脱” 就一个字

我们知道，整个故事里一直围绕着 QH 的队列在说来说去，不停地进行着队列操作，有时候把 QH 连接（link）起来成一个个的队列，而有时候又把 QH 从队列里给“unlink”，unlink 翻译成中文就是解开，拆开，松开。Okay，简洁一点说，一个字，脱！

还记得 `skel_unlink_qh` 么？`skelqh[]` 数组里边 11 个元素，另外那 10 个都知道怎么回事了，但是第 1 个元素，或者说这 0 号元素，一直就不太明白。现在就来仔细解读一下。

事实上，`uhci_unlink_qh` 函数有这么一句话：`list_move_tail(&qh->node, &uhci->skel_unlink_qh->node)`，换言之，凡是调用过 `uhci_unlink_qh` 的 QH，最终都被加入到了由 `skel_unlink_qh` 领衔的队列。但问题是加入这个队列之后呢？是不是就算隐退江湖了？其实不然，生活哪有那么简单啊？不是想退出江湖就能退出江湖的。咱们回过头来看这个函数，`uhci_scan_schedule`:



```

1708 static void uhci_scan_schedule(struct uhci_hcd *uhci)
1709 {
1710     int i;
1711     struct uhci_qh *qh;
1712
1713     /* Don't allow re-entrant calls */
1714     if (uhci->scan_in_progress) {
1715         uhci->need_rescan = 1;
1716         return;
1717     }
1718     uhci->scan_in_progress = 1;
1719 rescan:
1720     uhci->need_rescan = 0;
1721     uhci->fsbr_is_wanted = 0;
1722
1723     uhci_clear_next_interrupt(uhci);
1724     uhci_get_current_frame_number(uhci);
1725     uhci->cur_iso_frame = uhci->frame_number;
1726
1727     /* Go through all the QH queues and process the URBs in each one */
1728     for (i = 0; i < UHCI_NUM_SKELOH - 1; ++i) {
1729         uhci->next_qh = list_entry(uhci->skelqh[i]->node.next,
1730                                   struct uhci_qh, node);
1731         while ((qh = uhci->next_qh) != uhci->skelqh[i]) {
1732             uhci->next_qh = list_entry(qh->node.next,
1733                                       struct uhci_qh, node);
1734
1735             if (uhci_advance_check(uhci, qh)) {
1736                 uhci_scan_qh(uhci, qh);
1737                 if (qh->state == QH_STATE_ACTIVE) {
1738                     uhci_urbp_wants_fsbr(uhci,
1739                                           list_entry(qh->queue.next, struct urb_priv, node));
1740                 }
1741             }
1742         }
1743     }
1744
1745     uhci->last_iso_frame = uhci->cur_iso_frame;
1746     if (uhci->need_rescan)
1747         goto rescan;
1748     uhci->scan_in_progress = 0;
1749
1750     if (uhci->fsbr_is_on && !uhci->fsbr_is_wanted &&
1751         !uhci->fsbr_expiring) {
1752         uhci->fsbr_expiring = 1;
1753         mod_timer(&uhci->fsbr_timer, jiffies + FSBR_OFF_DELAY);
1754     }
1755
1756     if (list_empty(&uhci->skel_unlink_qh->node))
1757         uhci_clear_next_interrupt(uhci);
1758     else
1759         uhci_set_next_interrupt(uhci);
1760 }

```

1728 行的 for 循环，对 skelqh[] 数组从 0 开始循环，直到 9。1 到 9 就不说了，而顺着 0 往下看，针对 skel\_unlink\_qh 队伍里的每一个 qh 进行循环，每一个 qh 执行一次 uhci\_advance\_check()，而 skel\_unlink\_qh 这个队伍里的 qh 有一部分是上一次刚刚在

uhci\_advance\_check 中设置了 wait\_expired 为 1 的，另一部分可能没有设置过，因为调用 uhci\_unlink\_qh() 的并非只有 uhci\_advance\_check()，还有别的地方。而别的地方调用它的话就和超时不超时没有关系了。

于是现在分两种情况来看待这个 uhci\_advance\_check。第一种，qh 是因为超时被拉进 skel\_unlink\_qh 的，那么 1673 行 if 条件是满足的，这种情况下 uhci\_advance\_check 就直接返回了，但是返回的肯定是 1。返回了之后来到 uhci\_scan\_schedule，1736 行，uhci\_scan\_qh 就会被执行，进入到 uhci\_scan\_qh 中，1602 行，由于 qh 中还有 urb，1617 行的 uhci\_activate\_qh 就会被执行，因此 qh 将重新激活，qh->state 会被设置为 QH\_STATE\_ACTIVE，它会再次被拉入它自己的归属。于是又幸运地获得了重生。

而对于第二种情况，也是通过 uhci\_unlink\_qh() 给拉入 skel\_unlink\_qh 了。但是人家起码没超时，所以这次再看 uhci\_advance\_check，1673 行这个 if 条件就不一定满足了。然后如果真没超时，那么 1697 行会被执行，而 1697 这个 if 条件是否满足得看 1654 行这个 if 是否满足了，如果 1654 行满足，换言之，qh->state 不是 QH\_STATE\_ACTIVE，则设置 urbp 为空，而我们知道 uhci\_unlink\_qh 会把 qh->state 设置为 QH\_STATE\_UNLINKING，所以，1654 行肯定满足。而 urbp 设置为了 NULL，因此 1697 行这个 if 不会满足。因此，对于 unlink 的 qh，uhci\_advance\_check 这次除了返回 1 其他什么也不做，但是回到了 uhci\_scan\_schedule 之后，uhci\_scan\_qh 会执行，1622 行。

```
1532 #define QH_FINISHED_UNLINKING(qh) \
1533     (qh->state == QH_STATE_UNLINKING && \
1534      uhci->frame_number + uhci->is_stopped != qh->unlink_frame)
```

首先 qh->unlink\_frame 当初是在 uhci\_unlink\_qh() 中设置的，设置的就是当时的 Frame 号。而 uhci->frame\_number 是当前的 Frame 号。但对于眼前这个宏的含义，我曾经一度困惑过。我猜测这个宏判断的就是一个 QH 是否已经彻底失去利用价值，但我并不清楚为什么这个宏被这样定义。后来，Alan Stern 语重心长地说：

“When a QH is unlinked, the controller is allowed to continue using it until the end of the frame. So the unlink isn't finished until the frame is over and a new frame has begun. qh->unlink\_frame is the frame number when the QH was unlinked. uhci->frame\_number is the current frame number. If the two are unequal then the unlink is finished.”

没错，当一个 QH 在某个 Frame 被“unlink”了之后，在这个 Frame 结束之后主机控制器就不会再使用它了。也就是说，到下一个 Frame 开始，这个 QH 就算是真正地完成了它的“脱”。

这里尤其需要注意的是 uhci->is\_stopped，顾名思义，当 UHCI 正常工作时这个值应该为 0，而只有 UHCI 停了下来时，这个值才会是非 0。但我们知道，如果主机控制器自己都停止了下來，那么显然这个 QH 就算是彻底“脱”了，因为主机控制器不可能再使用它了，或者说主机

控制器不可能再访问它了，而停下来了就意味着 `is_stopped` 不为 0，显然，只要 `is_stopped` 不为 0，则 `uhci->frame_number+uhci->is_stopped` 是不可能等于 `qh->unlink_frame` 的。（`uhci->frame_number>=qh->unlink_frame` 恒成立，而 `is_stopped` 事实上永远大于等于 0。）

所以，1622 行这个宏这么一判断，发现 QH 确实已经没有利用价值了，就调用 `uhci_make_qh_idle` 从而把 `qh->state` 设置为 `QH_STATE_IDLE`，并且把本 QH 拉入 `uhci->idle_qh_list`，一旦加入这个 list，这个 QH 将从此“永不见天日”，没有人会再去理睬它了。不过，最后，关于 `uhci_scan_qh`，有三点需要强调一下。

第一点，1569 行调用了 `uhci_giveback_urb()`，为何在 1594 行也调用了 `uhci_giveback_urb`。但事实上你会发现任何一个 urb 都不可能在这两处先后被调用，要么在前者被调用，要么在后者被调用。道理很简单，一旦调用了 `uhci_giveback_urb()`，那么其 `urbp` 就会脱离 QH 的队列。这是 `uhci_giveback_urb()` 中 1507 行 `list_del_init` 干的。甚至 `urbp` 的内存也会被释放掉，这是 `uhci_giveback_urb()` 中 1514 行 `uhci_free_urb_priv()` 函数干的。

所以，事实上，对于大多数正常工作正常结束的 urb，在 `uhci_scan_qh` 中，1569 行这个 `uhci_giveback_urb` 会被调用，而一旦调用了，这个 `urbp` 就不复存在了，因此之后的代码就跟它毫无关系了。

那么另一方面，1551 行和 1571 行这两个 `break` 语句的存在使得 `while` 循环有可能提前结束，这就意味着，`while` 循环结束时，`qh->queue` 里面的 `urbp` 并不一定全部被遍历到了，因此，也就是说有些 urb 可能并没有执行 1569 行的 `uhci_giveback_urb()`，因此它们就有可能在 1594 行被传递给 `uhci_giveback_urb`。

第二点，1595 行 `goto restart` 是什么意思？乍一看，我愣是以为这个 `goto restart` 会导致这段代码成为死循环，可后来我算是琢磨出来了，`list_for_each_entry` 不是想遍历 `qh->queue` 这个 `urbp` 构成的队列么？可是每次如果它走到 1594 行这个 `uhci_giveback_urb` 的话，该 `urbp` 会被删掉，于是队列就改变了，好家伙，你在调用 `list_for_each_entry` 遍历队列时改变队列那还能不出事？所以也就别犹豫了，重新调用 `list_for_each_entry`，重新遍历不就成了么？

第三点，虽然 `uhci_scan_qh` 函数看上去挺复杂，但是正如 1574 行这个注释所说的那样，事实上对于大多数情况来说，这个函数执行到 1579 行就会返回。只有两种情况下才会执行 1579 之后的代码，第一个就是 1576 行 `QH_FINISHED_UNLINKING` 条件满足，即这个 QH 是刚刚被“unlink”，刚刚完成“脱”的，这种情况下要继续往下走，第二个就是虽然不是完成了“脱”的，但 `is_stopped` 不为 0。

但是虽然说这两种情况是小概率事件，但毕竟这节讨论的就是 `QH_FINISHED_UNLINKING`，所以这种情况究竟怎么处理还是要关注的。而这其中除了 1623 行 `uhci_make_qh_idle` 是最后一步要做的事情之外，1590 行，`uhci_cleanup_queue` 也没有仔细看过。既然整个故事已经进入了 `cleanup` 的阶段，那么就以这个 `cleanup` 函数作为结束吧。它来自 `drivers/usb/host/uhci-q.c`：

```

319 static int uhci_cleanup_queue(struct uhci_hcd *uhci, struct uhci_qh *qh,
320                             struct urb *urb)
321 {
322     struct urb_priv *urbp = urb->hcpriv;
323     struct uhci_td *td;
324     int ret = 1;
325
326     /* Isochronous pipes don't use toggles and their TD link pointers
327      * get adjusted during uhci_urb_dequeue(). But since their queues
328      * cannot truly be stopped, we have to watch out for dequeues
329      * occurring after the nominal unlink frame. */
330     if (qh->type == USB_ENDPOINT_XFER_ISOC) {
331         ret = (uhci->frame_number + uhci->is_stopped !=
332              qh->unlink_frame);
333     goto done;
334     }
335
336     /* If the URB isn't first on its queue, adjust the link pointer
337      * of the last TD in the previous URB. The toggle doesn't need
338      * to be saved since this URB can't be executing yet. */
339     if (qh->queue.next != &urbp->node) {
340         struct urb_priv *purbp;
341         struct uhci_td *ptd;
342
343         purbp = list_entry(urbp->node.prev, struct urb_priv, node);
344         WARN_ON(list_empty(&purbp->td_list));
345         ptd = list_entry(purbp->td_list.prev, struct uhci_td,
346                        list);
347         td = list_entry(urbp->td_list.prev, struct uhci_td,
348                        list);
349         ptd->link = td->link;
350         goto done;
351     }
352
353     /* If the QH element pointer is UHCI_PTR_TERM then then currently
354      * executing URB has already been unlinked, so this one isn't it. */
355     if (qh_element(qh) == UHCI_PTR_TERM)
356         goto done;
357     qh->element = UHCI_PTR_TERM;
358
359     /* Control pipes don't have to worry about toggles */
360     if (qh->type == USB_ENDPOINT_XFER_CONTROL)
361         goto done;
362
363     /* Save the next toggle value */
364     WARN_ON(list_empty(&urbp->td_list));
365     td = list_entry(urbp->td_list.next, struct uhci_td, list);
366     qh->needs_fixup = 1;
367     qh->initial_toggle = uhci_toggle(td_token(td));
368
369 done:
370     return ret;
371 }

```

首先注释里说得也很清楚。330 行，对于 ISO 类型，它并不使用 toggle bits，所以这里就是判断是否彻底“脱”了，是就返回 1。

339 行，如果当前讨论的这个 urb 不是 `qh->queue` 队列里的第一个 urb，那么就进入 if 里面的语句，`purbp` 将是 `urbp` 的前一个节点，即前一个 urb，`ptd` 则是 `purbp` 的 TD 队列中最后一个 TD，而 `td` 又是 `urbp` 的 TD 队列中最后一个 TD。让 `ptd` 的 `link` 指向 `td` 的 `link`，即让前一个 urb 的最后一个 TD 指向原来 urb 的最后一个 TD 所指向的位置。有了“接班人”之后，当前这个 urb 或者说这个 urb 就可以淡出“历史舞台”了。

357 行，让 `qh->element` 等于 `UHCI_PTR_TERM`，等于宣布 QH 正式“退休”。

365 行、366 行、367 行的目的也很明确，保存好下一个 TD 的 `toggle` 位，以待时机进行修复。至于如何修复，在讲 `uhci_giveback_urb` 和 `uhci_scan_qh` 中都已经看到了，会通过判断 `needs_fixup` 来执行相应的代码。此处不再赘述。

关于“脱”，就讲到这里吧。

## 第 4 篇

# Linux 那些事儿之我是 U 盘

1. 小城故事.....	388	22. 通往春天的管道.....	436
2. Makefile.....	389	23. 传说中的 URB.....	440
3. 变态的模块机制.....	390	24. 彼岸花的传说（一）.....	443
4. 想到达明天现在就要启程.....	392	25. 彼岸花的传说（二）.....	445
5. 外面的世界很精彩.....	394	26. 彼岸花的传说（三）.....	448
6. 未曾开始却似结束.....	395	27. 彼岸花的传说（四）.....	451
7. 狂欢是一群人的孤单.....	396	28. 彼岸花的传说（五）.....	453
8. 总线、设备和驱动（上）.....	397	29. 彼岸花的传说（六）.....	457
9. 总线、设备和驱动（下）.....	398	30. 彼岸花的传说（七）.....	460
10. 我是谁的他.....	400	31. 彼岸花的传说（八）.....	463
11. 从协议中来，到协议中去（上）.....	401	32. 彼岸花的传说（The End）.....	467
12. 从协议中来，到协议中去（中）.....	403	33. SCSI 命令之我型我秀.....	468
13. 从协议中来，到协议中去（下）.....	405	34. 迷雾重重的批量传输（一）.....	472
14. 梦开始的地方.....	406	35. 迷雾重重的批量传输（二）.....	476
15. 设备花名册.....	411	36. 迷雾重重的批量传输（三）.....	479
16. 冰冻三尺非一日之寒.....	412	37. 迷雾重重的批量传输（四）.....	484
17. 冬天来了，春天还会 远吗？（一）.....	416	38. 迷雾重重的批量传输（五）.....	489
18. 冬天来了，春天还会 远吗？（二）.....	422	39. 迷雾重重的批量传输（六）.....	493
19. 冬天来了，春天还会 远吗？（三）.....	425	40. 迷雾重重的批量传输（七）.....	495
20. 冬天来了，春天还会 远吗？（四）.....	427	41. 跟着感觉走（一）.....	500
21. 冬天来了，春天还会 远吗？（五）.....	431	42. 跟着感觉走（二）.....	503
		43. 有多少爱可以胡来？（一）.....	509
		44. 有多少爱可以胡来？（二）.....	513
		45. 当梦醒了天晴了.....	518
		46. 其实世上本有路，走的人多了， 即便没了路.....	522

## 1. 小城故事

这个故事中使用的是 2.6.22 的内核代码。在 Linux 内核代码目录中，所有与设备驱动程序有关的代码都在 `drivers/` 目录下面，在这个目录中用 `ls` 命令可以看到很多子目录：

```
lfg1:/usr/src/linux-2.6.22/drivers # ls
Kconfig  acpi  atm  block  char  cpufreq  dma  fc4  hid  ide
input  leds  md  mfd  mtd  oprofile  pci  ps3  s390  serial  spi
usb  zorro  Makefile  amba  auxdisplay  bluetooth  clocksource  crypto
edac  firewire  hwmon  ieee1394  isdn  macintosh  media  misc  net
parisc  pcmcia  rapidio  sbus  sh  tc  video  acorn  ata
base  cdrom  connector  dio  eisa  firmware  i2c  infiniband
kvm  mca  message  mmc  nubus  parport  pnp  rtc  scsi  sn
telephony  wl
```

其中 `usb` 目录包含了所有 USB 设备的驱动，而 `usb` 目录下面又有它自己的子目录，进去看一下：

```
lfg1:/usr/src/linux-2.6.22/drivers # cd usb/
lfg1:/usr/src/linux-2.6.22/drivers/usb # ls
Kconfig  Makefile  README  atm  class  core  gadget  host  image  misc  mon
serial  storage  usb-skeleton.c
```

注意到每一个目录下面都有一个 `Kconfig` 文件和 `Makefile`，这很重要。稍后会有介绍。

而我们的故事其实是围绕着 `drivers/usb/storage` 这个目录来展开的。实际上这里的代码清清楚楚地展示了我们日常频繁接触的 U 盘是如何工作的，是如何被驱动起来的。但是这个目录里边的代码并不是生活在世外桃源，它们总是和外面的世界有着千丝万缕的瓜葛。可以继续进来看一下：

```
lfg1:/usr/src/linux-2.6.22/drivers/usb # cd storage/
lfg1:/usr/src/linux-2.6.22/drivers/usb/storage # ls
Kconfig  alauda.h  debug.c  dpcm.h  initializers.c  isd200.h  karma.c
onetouch.c  protocol.h  sddr09.c  sddr55.h  transport.c  usb.c
Makefile  datafab.c  debug.h  freecom.c  initializers.h  jumpshot.c
karma.h  onetouch.h  scsiglue.c  sddr09.h  shuttle_usbat.c  transport.h
usb.h  alauda.c  datafab.h  dpcm.c  freecom.h  isd200.c  jumpshot.h
libusual.c  protocol.c  scsiglue.h  sddr55.c  shuttle_usbat.h
unusual_devs.h
```

乍一看，着实吓了一跳，用 `wc -l *` 这个命令统计一下，15455 行!!! 但是，也许，生活中总是充满了跌宕起伏。

认真看了一下 `Makefile` 和 `Kconfig` 之后，心情明显好了许多。

## 2. Makefile

基本上，Linux 内核中每一个目录下边都有一个 Makefile。Makefile 和 Kconfig 就像一个城市的地图，地图带领我们去认识一个城市，而 Makefile 和 Kconfig 则可以让我们了解这个目录下面的结构。drivers/usb/storage/目录下边的 Makefile 内容如下：

```
lfg1:/usr/src/linux-2.6.22/drivers/usb/storage # cat Makefile
#
# Makefile for the USB Mass Storage device drivers.
#
# 15 Aug 2000, Christoph Hellwig <hch@infradead.org>
# Rewritten to use lists instead of if-statements.
#

EXTRA_CFLAGS      := -Idrivers/scsi

obj-$(CONFIG_USB_STORAGE)      += usb-storage.o

usb-storage-obj-$(CONFIG_USB_STORAGE_DEBUG)      += debug.o
usb-storage-obj-$(CONFIG_USB_STORAGE_USBAT)      += shuttle_usbat.o
usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR09)      += sddr09.o
usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR55)      += sddr55.o
usb-storage-obj-$(CONFIG_USB_STORAGE_FREECOM)      += freecom.o
usb-storage-obj-$(CONFIG_USB_STORAGE_DPCM)      += dpcm.o
usb-storage-obj-$(CONFIG_USB_STORAGE_ISD200)      += isd200.o
usb-storage-obj-$(CONFIG_USB_STORAGE_DATAFAB)      += datafab.o
usb-storage-obj-$(CONFIG_USB_STORAGE_JUMPSHOT)      += jumpshot.o
usb-storage-obj-$(CONFIG_USB_STORAGE_ALAUDA)      += alauda.o
usb-storage-obj-$(CONFIG_USB_STORAGE_ONETOUCH)      += onetouch.o
usb-storage-obj-$(CONFIG_USB_STORAGE_KARMA)      += karma.o

usb-storage-objs :=      scsiglue.o protocol.o transport.o usb.o \
                        initializers.o $(usb-storage-obj-y)

ifneq ($(CONFIG_USB_LIBUSUAL),)
    obj-$(CONFIG_USB)      += libusual.o
endif
```

而 Kconfig 文件，其实就是对上面看到的这些 Config 选项进行解释，Kconfig 文件比较长，就不贴出来了。但是通过看 Kconfig 文件可以知道，除了 CONFIG\_USB\_STORAGE 这个编译选项是我们真正需要的以外，别的选项都可以不予理睬。比如，关于 CONFIG\_USB\_STORAGE\_DATAFAB，Kconfig 文件中有这么一段，

```
config USB_STORAGE_DATAFAB
    bool "Datafab Compact Flash Reader support (EXPERIMENTAL)"
    depends on USB_STORAGE && EXPERIMENTAL
    help
        Support for certain Datafab CompactFlash readers.
        Datafab has a web page at <http://www.datafabusa.com/>.
```

显然，这个选项和我们没有关系，首先这是专门针对 Datafab 公司的产品的，其次 CompactFlash reader 是一种 flash 设备，但这显然不是 U 盘，因为 drivers/usb/storage 这个目录



里面的代码是针对一类设备的，不是某一种特定的设备，这一类设备就是 USB Mass Storage 设备。关于这类设备，有专门的文档进行介绍，有相应的 spec 描述这类设备的通信或者物理上电特性上等方面的规范，U 盘只是其中的一种，这种设备使用的通信协议被称为 Bulk-Only Transport 协议。再比如，关于 CONFIG\_USB\_STORAGE\_SDDR55 这个选项，Kconfig 文件中也有对应的一段：

```
config USB_STORAGE_SDDR55
    bool "SanDisk SDDR-55 SmartMedia support (EXPERIMENTAL)"
    depends on USB_STORAGE && EXPERIMENTAL
    help
        Say Y here to include additional code to support the Sandisk SDDR-55
        SmartMedia reader in the USB Mass Storage driver.
```

很显然这是 SanDisk 的产品，并且是针对 SM 卡的，这也不是 U 盘，所以都不去理睬了。事实上，很容易确定，只有 CONFIG\_USB\_STORAGE 这个选项是需要真正关心的，而它所对应的模块叫 usb-storage，Makefile 中最后几行也说了：

```
usb-storage-objs :=      scsiglue.o protocol.o transport.o usb.o \
                        initializers.o $(usb-storage-obj-y)
```

这就意味着我们只需要关注的文件就是 scsiglue.c, protocol.c, transport.c, usb.c, initializers.c 以及它们同名的.h 头文件。再次使用 wc -l 命令统计一下这几个文件，发现总长度只有 3654 行，比最初看到的 15455 多行少了许多，当时就信心倍增。

不过需要特别注意的是，CONFIG\_USB\_STORAGE\_DEBUG 这个编译选项不是必需的，但是如果真的要自己修改或者调试 usb-storage 的代码，那么打开这个选项是很有必要的，因为它会负责打印一些调试信息，以后在源代码中会看到它的作用。

---

## 3. 变态的模块机制

有一种感动，叫泪流满面，有一种机制，叫模块机制，“十月革命”一声炮响，给 Linux 送来了模块机制。显然，这种模块机制给那些 Linux 的发烧友们带来了方便，因为模块机制意味着人们可以把庞大的 Linux 内核划分为许许多多多个小的模块，对于编写设备驱动程序的那帮家伙来说，从此以后他们可以编写设备驱动程序却不需要把她编译进内核，不用 reboot 机器。她只是一个模块，当你需要她时，你可以把她“抱入怀中”(insmod)，当你不再需要她时，你可以把她一脚踢开，甚至，你可以对她咆哮：“滚吧！”(rmmod)。

于是，忽如一夜春风来，内核处处是模块。让我们从一个伟大的例子去认识模块。这就是传说中的“Hello World!”，这个梦幻般的名字看过无数次了，每一次她出现在眼前，就意味着我们开始接触一种新的计算机语言了，或者，如此刻，开始描述一个新的故事。

请看下面这段代码，她就是 Linux 下的一个最简单的模块。当你安装这个模块时，她会用她特有的语言向你表白：“Hello, world!”，而后来你卸载了这个模块，无情抛弃了她，她很伤心，她很绝望，但她没有抱怨，她只是淡淡地说，“Goodbye, cruel world!”（再见，残酷的世界!）

```

/***** hello.c *****/
1 #include <linux/init.h> /* Needed for the macros */
2 #include <linux/module.h> /* Needed for all modules */
3 MODULE_LICENSE("Dual BSD/GPL");
4 MODULE_AUTHOR("fudan_abc");
5
6 static int __init hello_init(void)
7 {
8     printk(KERN_ALERT "Hello, world!\n");
9     return 0;
10 }
11
12 static void __exit hello_exit(void)
13 {
14     printk(KERN_ALERT "Goodbye, cruel world\n");
15 }
16
17 module_init(hello_init);
18 module_exit(hello_exit);

```

需要使用 `module_init()` 和 `module_exit()`，你可以称之为函数，不过实际上它们是一些宏，现在你可以不用去知道它们背后的故事，只需要知道，在 2.6 内核的世界里，你写的任何一个模块都需要使用它们来初始化或退出，或者说注册以及后来的注销。当你用 `module_init()` 为一个模块注册了之后，在使用 `insmod` 这个命令去安装时，`module_init()` 注册的函数将会被执行。而当你用 `rmmod` 这个命令去卸载一个模块时，`module_exit()` 注册的函数将会被执行。`module_init()` 被称为驱动程序初始化入口（Driver Initialization Entry Point）。怎么样演示以上代码的运行呢？没错，你需要一个 Makefile。

```

1 # To build modules outside of the kernel tree, we run "make"
2 # in the kernel source tree; the Makefile these then includes this
3 # Makefile once again.
4 # This conditional selects whether we are being included from the
5 # kernel Makefile or not.
6 ifeq ($(KERNELRELEASE),)
7
8     # Assume the source tree is where the running kernel was built
9     # You should set KERNELDIR in the environment if it's elsewhere
10    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
11    # The current directory is passed to sub-makes as argument
12    PWD := $(shell pwd)
13
14 modules:
15     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
16
17 modules_install:
18     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
19

```

```
20 clean:
21     rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .tmp_versions
22
23 .PHONY: modules modules_install clean
24
25 else
26     # called from kernel build system: just declare what our modules are
27     obj-m := hello.o
28 endif
```

因为 2.6 内核要求你在编译模块之前，必须先在内核源代码目录下执行“make”，换言之，你必须先配置过内核，执行过“make”，然后才能“make”你自己的模块。原因我就不细说了，你按着她要求的这么去做就行了。在内核顶层目录“make”过之后，你就可以在你当前放置 Makefile 的目录下执行“make”了。“make”之后你就应该看到一个叫做 hello.ko 的文件生成了。恭喜你！这就是你将要测试的模块。

执行命令：

```
#insmod hello.ko
```

同时在另一个窗口，用命令 `tail -f /var/log/messages` 查看日志文件，你会看到 Hello world 被打印了出来。再执行命令：

```
#rmmod hello.ko
```

此时，在另一窗口你会看到“Goodbye, cruel world!”被打印了出来。

到这里，我该恭喜你，因为你已经能够编写 Linux 内核模块了。这种感觉很美妙，不是吗？你可以嘲笑秦皇汉武略抒文采唐宗宋祖稍领风骚，还可以嘲笑一代天骄成吉思汗只识弯弓射大雕了。

日后我们会看到，2.6 内核中，每个模块都是以 `module_init` 开始，以 `module_exit` 结束。对大多数人来说没有必要知道这是为什么，记住就可以了，相信每一个对 Linux 有一点常识的人都会知道这一点的。

---

## 4. 想到达明天现在就要启程

既然知道了编写模块的方法，那么编写设备驱动程序自然也就不难了。我相信，每一个会写模块的人都不会觉得写设备驱动有困难。

真的，我没说假话，写驱动不是什么难事，你完全可以很自信地说，你已经可以写设备驱动了。对，没错，“飘柔，就这么自信。”

前面说了每一个模块都是以 `module_init` 开始，以 `module_exit` 结束的，那么我们就来看一下 U 盘驱动的这个模块。在茫茫人海中，我们很容易找到这个文件 `drivers/usb/storage/usb.c`，在这个文件中又不难发现下面这段：

```

1068 static int __init usb_stor_init(void)
1069 {
1070     int retval;
1071     printk(KERN_INFO "Initializing USB Mass Storage driver...\n");
1072
1073     /* register the driver, return usb_register return code if error */
1074     retval = usb_register(&usb_storage_driver);
1075     if (retval == 0) {
1076         printk(KERN_INFO "USB Mass Storage support registered.\n");
1077         usb_usual_set_present(USB_US_TYPE_STOR);
1078     }
1079     return retval;
1080 }
1081
1082 static void __exit usb_stor_exit(void)
1083 {
1084     US_DEBUGP("usb_stor_exit() called\n");
1085
1086     /* Deregister the driver
1087      * This will cause disconnect() to be called for each
1088      * attached unit
1089      */
1090     US_DEBUGP("-- calling usb_deregister()\n");
1091     usb_deregister(&usb_storage_driver);
1092
1093     /* Don't return until all of our control and scanning threads
1094      * have exited. Since each thread signals threads_gone as its
1095      * last act, we have to call wait_for_completion the right number
1096      * of times.
1097      */
1098     while (atomic_read(&total_threads) > 0) {
1099         wait_for_completion(&threads_gone);
1100         atomic_dec(&total_threads);
1101     }
1102
1103     usb_usual_clear_present(USB_US_TYPE_STOR);
1104 }
1105
1106 module_init(usb_stor_init);
1107 module_exit(usb_stor_exit);

```

其实，`module_init/module_exit` 只是一个宏，通常写模块的人为了彰显自己的个性，会给自己的初始化函数和注销函数另外取名字，比如这里 `module_init(usb_stor_init)` 以及 `module_exit(usb_stor_exit)` 实际上就是告诉这个世界，真正的函数是 `usb_stor_init` 和 `usb_stor_exit`。这种伎俩在 Linux 内核代码中屡见不鲜，见多了也就不必大惊小怪了，天要下雨娘要嫁人，随她去吧。下面就从 `usb_stor_init` 正式开始我们的探索之旅。

## 5. 外面的世界很精彩

看代码之前，我曾经认真地思考过这么一个问题，需要关注的仅仅是 `drivers/usb/storage/` 目录下面那相关的 3000 多行代码吗？就是这样几个文件就能让一个个不同的 U 盘在 Linux 下面工作起来吗？像一开始那样把这个目录比成一个小城的话，也许，城里的月光很漂亮，它能够把人的梦照亮，能够温暖人的心房。但我们真的就能厮守在这个城里，一生一世吗？

很不幸，问题远不是这样简单。外面的世界很精彩，作为 U 盘，她需要与 USB Core、SCSI Core、内存管理单元，还有内核中许许多多其他模块打交道。外面的世界很大，远比我们想象的大。

什么是 USB Core？它负责实现一些核心的功能，为别的设备驱动程序提供服务，比如申请内存，比如实现一些所有设备都会需要的公共函数。事实上，在 USB 的世界里，要使一个普通的设备正常地工作，除了要有设备本身以外，还要有主机控制器（Host Controller），和这个控制器相连接在一起的是 Root Hub。

Hub 应该不会陌生，以太网上的 Hub 允许多个人的计算机共用一个网口。而 USB 的世界里同样有 Hub，其实原理是一样的，任何支持 USB 的电脑不会说只允许只能一个时刻使用一个 USB 设备，比如插入了 U 盘，同样还可以插入 USB 键盘，还可以再插一个 USB 鼠标，因为你会发现你的电脑里并不只是一个 USB 接口。这些口实际上就是所谓的 Hub 口。而现实中经常是让一个 USB 控制器和一个 Hub 绑定在一起，即集成，而这个 Hub 也被称作 Root Hub。换言之，和 USB 控制器绑定在一起的 Hub 就是系统中最根本的 Hub，其他 Hub 可以连接到她这里，然后可以延伸出去，外接别的设备，当然也可以不用别的 Hub，让 USB 设备直接接到 Root Hub 上。

那么 USB 主机控制器本身是干什么用的呢？控制器，顾名思义，用于控制所有 USB 设备的通信。通常计算机的 CPU 并不是直接和 USB 设备打交道，而是和控制器打交道，它要对设备做什么，他会告诉控制器，而不是直接把指令发给设备，控制器再去负责处理这件事情，会去指挥设备执行命令，而 CPU 就不用管剩下的事情，控制器会替他去完成剩下的事情，事情办完了再通知 CPU。

否则让 CPU 去盯着每一个设备做每一件事情，那是不现实的，那就好比让一个学院的院长去盯着我们每一个本科生上课，去管理我们的出勤，只能说，不现实。所以我们就被分成了几个系，通常院长有什么指示直接跟各系领导说就可以了，如果他要和三个系主任说事情，他即使不把三个人都召集起来开个会，也可以给三个人各打一个电话，打完电话他就忙他自己的事情去了。

所以，Linux 内核开发人员们，专门写了一些代码，并美其名曰 USB Core。时代总在发展，早期的 Linux 内核，其结构并不是如今天这般有层次感，远不像今天这般错落有致，那时候

drivers/usb/这个目录下边放了很多很多文件，USB Core 与其他各种设备的驱动程序的代码都堆砌在这里，后来，怎奈世间万千的变幻，总爱把有情的人分两端。

于是在 drivers/usb/目录下面出来了一个 core 目录，就专门放一些核心的代码，比如初始化整个 USB 系统，初始化 Root Hub，初始化主机控制器的代码，再后来甚至把主机控制器相关的代码也单独建了一个目录，叫 host 目录，这是因为 USB 主机控制器随着时代的发展，也开始有了好几种，不再像刚开始那样只有一种，所以呢，设计者们把一些主机控制器公共的代码仍然留在 core 目录下，而一些各主机控制器单独的代码则移到 host 目录下面让负责各种主机控制器的人去维护。常见的主机控制器有三种：EHCI、UHCI 和 OHCI。

所以这样，出来了三个概念，USB Core、USB 主机控制器和 USB 设备，现实总是很无奈，然而，若知道灵犀的方向，哪怕不能够朝夕相伴？没错，USB 通信的灵魂就是 USB 协议。USB 协议将是所有 USB 设备和 USB 主机所必须遵循的游戏规则。这种规则也很自然地体现在了代码中。于是，我们需要了解的不仅仅是 drivers/usb/storage/目录下面的文件，还得去了解外面的世界，虽然，只需要了解一点点。

---

## 6. 未曾开始却似结束

还是 usb\_stor\_init 这个初始化函数，看了它的代码，每一个人的心中都有一种莫名的兴奋，因为它太短了，就几行，除了两个 printk 语句以外，就是一个函数的调用，usb\_register。（另外还有一个函数，usb\_usual\_set\_present，但是该函数基本上与本故事无关，它和 usb\_stor\_exit 中调用的 usb\_usual\_clear\_present 是一对，都可以无视之。）

printk 不用我说，每一个有志青年都该知道，就算没见过 printk 也该见过 printf 吧，在谭浩强大哥的带领下我们学会了使用#include<stdio.h>->main()->printf()来打印“hello, world!”。而 stdio.h 就是一个 C 库，printf 是一个函数，来自函数库。可是内核中没有标准 C 库，开发人员自己准备了一些函数，专门用于内核代码中，所以就出来了一个 printk。printk 中的“k”就是 kernel，即内核。所以我们只要把它当做 printf 的兄弟即可。如果感兴趣，可以去研究一下 printk 的特点，它和 printf 多少有些不同，但基本思想是一样的。所以我们就不多讲了，当然驱动程序中所有的 printk 语句对 U 盘的工作都没有什么用，它无非是打出来给我们看的，或者打印给用户看，或者打印给开发人员看，特别是开发人员要调试程序时，就会很有用。

于是我们更开心了，不用看 printk，那就只有一个函数调用了，usb\_register。这个函数有什么用？首先这个函数正是来自 USB Core。凡是 USB 设备驱动，都要调用这个函数来向 USB Core 注册，从而让 USB Core 知道有这个设备。这就像政府规定一对夫妻结婚要到相关部门那里去登记是一样的，我们无需知道政府是如何管理的，只需要知道去政府那里登记即可。

这样，insmod 时，usb\_stor\_init 这个函数会被调用，初始化就算完成了。于是设备就开始工作了。而当我们 rmmod 时，usb\_stor\_exit 这个函数会被调用。我们发现，这个函数也很短，我们能看出来，US\_DEBUG 也就是打印一些调试信息。因此，这里实际上也就是调用了函数 usb\_deregister()，它和 usb\_register() 是一对，完成了注销的工作，从此设备就从 USB Core 中消失了。于是我们发现，编写设备驱动竟是如此简单，驱动程序真的就这么结束了？

这一切，不禁让人产生了一种幻觉，让人分不清故事从哪里开始，又从哪里结束，一切都太短暂了。仿佛开始在结束时开始，而结束却在开始时就早已结束。

真的吗？答案是否定的。

我们并无意去详细介绍 2.6 节内核中的设备模型，但是不懂设备模型又怎能说自己懂设备驱动呢？读代码的人，写代码的人，都要知道，什么是设备驱动？什么又是设备？设备和驱动之间究竟是什么关系？设备如何与计算机主机联系起来？我相信在中关村卖盗版光盘的哥们也能回答这个问题。计算机世界里，设备有很多种，比如 PCI 设备，比如 ISA 设备，再比如 SCSI 设备，再比如我们这里的 USB 设备。为设备联姻的是总线，是它把设备连入了计算机主机。但是与其说设备是嫁给了计算机主机，倒不如说设备是嫁给了设备驱动程序。很显然，在计算机世界里，无论风里雨里，陪伴着设备的正是驱动程序。

唯一的遗憾是，计算机中的设备和驱动程序的关系却并非如可乐和拉环的关系那样，一对一。然而世上又有多少事情总能如人愿呢。

---

## 7. 狂欢是一群人的孤单

Linux 设备模型中三个很重要的概念就是总线、设备和驱动，即 bus、device 和 driver。而实际上内核中也定义了这么一些数据结构，它们是 struct bus\_type，struct device，struct device\_driver，这三个重要的数据结构都来自同一个地方，即 include/linux/device.h。

我们知道总线有很多种，如 PCI 总线、SCSI 总线、USB 总线，所以我们会看到 Linux 内核代码中出现 pci\_bus\_type，scsi\_bus\_type，usb\_bus\_type，它们都是 struct bus\_type 类型的变量。而 struct bus\_type 结构中两个非常重要的成员就是 struct kset drivers 和 struct kset devices。kset 和另一个叫做 kobject 正是 2.6 内核中设备模型的基本元素。

这里我们只需要知道，drivers 和 devices 的存在，让 struct bus\_type 与两个链表联系了起来：分别是 devices 和 drivers 的链表。也就是说，知道一条总线所对应的数据结构，就可以找到这条总线所关联的设备，及支持这类设备的驱动程序。

而要实现这些目的，就要求每次出现一个设备就要向总线汇报，或者说注册。每次出现一个驱动，也要向总线汇报，或者注册。比如系统初始化时，会扫描连接了哪些设备，并为每一个设备建立起一个 `struct device` 的变量，每一次有一个驱动程序，就要准备一个 `struct device_driver` 结构的变量。把这些变量统统加入相应的链表，设备插入 `devices` 链表，驱动插入 `drivers` 链表。这样通过总线就能找到每一个设备、每一个驱动。

然而，假如计算机里只有设备却没有对应的驱动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。设备开始多了，驱动开始多了。它们像是来自两个世界，设备们彼此取暖，驱动们一起狂欢，但它们有一点是相同的，都只是在等待属于自己的那个另一半。

---

## 8. 总线、设备和驱动（上）

`struct bus_type` 中为设备和驱动准备了两个链表，而代表设备的结构体 `struct device` 中又有两个成员，`struct bus_type *bus` 和 `struct device_driver *driver`。同样，代表驱动的结构体 `struct device_driver` 同样有两个成员，`struct bus_type *bus` 和 `struct list_head devices`，`struct device` 和 `struct device_driver` 的定义和 `struct bus_type` 一样，在 `include/linux/device.h` 中。凭直觉，可以知道，`struct device` 中 `bus` 记录的是这个设备连在哪条总线上，`driver` 记录的是这个设备用的是哪个驱动。反过来，`struct device_driver` 中的 `bus` 代表的也是这个驱动属于哪条总线，`devices` 记录的是这个驱动支持的哪些设备。没错，是 `devices`（复数），而不是 `device`（单数），因为一个驱动程序可以支持一个或多个设备，反过来一个设备则只会绑定给一个驱动程序。

于是我们想知道，关于 `bus`，关于 `device`，关于 `driver`，它们是如何建立联系的呢？换言之，这三个数据结构中的指针是如何被赋值的？绝对不可能发生的事情是，一旦为一条总线申请了一个 `struct bus_type` 的数据结构之后，它就知道它的 `devices` 链表和 `drivers` 链表会包含哪些东西，这些东西一定不会是先天就有的，只能是后来填进来的。

而具体到 USB 系统，完成这个工作的就是 USB Core。USB Core 的代码会进行整个 USB 系统的初始化，比如申请 `struct bus_type usb_bus_type`，然后会扫描 USB 总线，看线上连接了哪些 USB 设备。或者说 Root Hub 上连了哪些 USB 设备，比如说连接了一个 USB 键盘，那么就为它准备一个 `struct device`，根据它的实际情况，为这个 `struct device` 赋值，并插入 `devices` 链表中来。又比如 Root Hub 上连了一个普通的 Hub，那么除了要为这个 Hub 本身准备一个 `struct device` 以外，还得继续扫描看这个 Hub 上是否又连了别的设备，如果有的话继续重复之前的事情，这样一直进行下去，直到完成整个扫描，最终就把 `usb_bus_type` 中的 `devices` 链表给建立了起来。



那 `drivers` 链表呢？这个就不用 `bus` 方面主动了，而该由每一个驱动本身去 `bus` 上面登记，或者说挂牌。具体到 USB 系统，每一个 USB 设备的驱动程序都会有一个 `struct usb_driver` 结构体，其代码如下，来自 `include/linux/usb.h`：

```
833 struct usb_driver {
834     const char *name;
835
836     int (*probe) (struct usb_interface *intf,
837                  const struct usb_device_id *id);
838
839     void (*disconnect) (struct usb_interface *intf);
840
841     int (*ioctl) (struct usb_interface *intf, unsigned int code,
842                  void *buf);
843
844     int (*suspend) (struct usb_interface *intf, pm_message_t message);
845     int (*resume) (struct usb_interface *intf);
846
847     void (*pre_reset) (struct usb_interface *intf);
848     void (*post_reset) (struct usb_interface *intf);
849
850     const struct usb_device_id *id_table;
851
852     struct usb_dynids dynids;
853     struct usbdrv_wrap drvwrap;
854     unsigned int no_dynamic_id:1;
855     unsigned int supports_autosuspend:1;
856 };
857 #define to_usb_driver(d) \
    container_of(d, struct usb_driver, drvwrap.driver)
```

此刻我们只需注意到其中的 `struct device_driver driver` 这个成员，USB Core 为每一个设备驱动准备了一个函数，让它把这个 `struct device_driver driver` 插入到 `usb_bus_type` 中的 `drivers` 链表中去。而这个函数正是我们此前看到的 `usb_register`。而与之对应的 `usb_deregister` 函数所从事的正是与之相反的工作，把这个结构体从 `drivers` 链表中删除。

可以说，USB Core 的确是用心良苦，为每一个 USB 设备驱动做足了功课，正因为如此，作为一个实际的 USB 设备驱动，它在初始化阶段所要做的事情就很少，很简单了，直接调用 `usb_register` 即可。事实上，没有人是理所当然应该为你做什么的，但 USB Core 这么做了。所以每一个写 USB 设备驱动的人应该铭记，USB 设备驱动绝不是一个人在工作，在它身后，是 USB Core 所提供的默默无闻又不可或缺的支持。

---

## 9. 总线、设备和驱动（下）

`bus` 上的两张链表记录了每一个设备和驱动，那么设备和驱动这两者之间又是如何联系起

来的呢？此刻，必须抛出这样一个问题，先有设备还是先有驱动？

在以前，先有的是设备，每一个要用的设备在计算机启动之前就已经插好了，插放在它应该的位置上，然后计算机启动，然后操作系统开始初始化，总线开始扫描设备。每找到一个设备，就为其申请一个 `struct device` 结构，并且挂入总线中的 `devices` 链表中，然后每一个驱动程序开始初始化，开始注册其 `struct device_driver` 结构，然后它去总线的 `devices` 链表去寻找（遍历），去寻找每一个还没有绑定驱动的设备，即 `struct device` 中的 `struct device_driver` 指针仍为空的设备，然后它会去观察这种设备的特征，看是否是它所支持的设备，如果是，那么调用一个叫做 `device_bind_driver` 的函数，然后它们就结为了秦晋之好。换句话说，把 `struct device` 中的 `struct device_driver` 指向这个驱动，而 `struct device_driver` 把 `struct device` 加入它的 `struct list_head devices` 链表中来。就这样，`bus`、`device` 和 `driver`，这三者之间或者说它们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个。一荣俱荣，一损俱损。

但现在情况变了，出现了一种新的名词“热插拔”。设备可以在计算机启动以后在插入或者拔出计算机了。因此，很难再说是先有设备还是先有驱动了。因为都有可能。设备可以在任何时刻出现，而驱动也可以在任何时刻被加载。所以，出现的情况就是，每当一个 `struct device` 诞生，它就会去 `bus` 的 `drivers` 链表寻找自己的另一半；反之，每当一个 `struct device_driver` 诞生，它就去 `bus` 的 `devices` 链表寻找它的那些设备。如果找到了合适的，那么和之前那种情况一样，调用 `device_bind_driver` 绑定好。如果找不到，没有关系，等待吧。

事实上，完善这个三角关系，正是每一个设备驱动初始化阶段所完成的重要使命之一。让我们还是回到代码中来，`usb_register` 函数调用是调用了，但是传递给它的参数是什么呢？

我们注意到，调用 `usb_register` 函数的代码如下：

```
1073      /* register the driver, return usb_register return code if error */
1074      retval = usb_register(&usb_storage_driver);
```

是的，传递了一个叫做 `usb_storage_driver` 的家伙，这是什么？同一个文件中：

```
1055 static struct usb_driver usb_storage_driver = {
1056     .name =          "usb-storage",
1057     .probe =         storage_probe,
1058     .disconnect =    storage_disconnect,
1059 #ifdef CONFIG_PM
1060     .suspend =       storage_suspend,
1061     .resume =        storage_resume,
1062 #endif
1063     .pre_reset =     storage_pre_reset,
1064     .post_reset =    storage_post_reset,
1065     .id_table =      storage_usb_ids,
1066 };
```

可以看到这定义了一个 `struct usb_driver` 的结构体变量，`usb_storage_driver`，关于 `usb_driver` 我们上节已经说过了，当时主要说的是其中的成员 `driver`，而眼下要讲的则是另外几个成员。

首先, `owner` 是用来给模块计数的, 每个模块都这么用, 赋值总是 `THIS_MODULE`。而 `name` 就是这个模块的名字, USB Core 会处理它, 所以如果这个模块正常被加载了的话, 使用 `lsmod` 命令能看到一个叫做 `usb-storage` 的模块名。CONFIG\_PM 是与电源管理相关的。下面重点要讲一讲, `.probe` 和 `.disconnect` 以及这个 `id_table`。

## 10. 我是谁的他

`probe`, `disconnect`, `id_table`, 这三个元素中首先要登场亮相的是 `id_table`, 它是干什么用的呢?

我们说过, 一个设备只能绑定一个驱动, 但驱动并非只能支持一种设备。道理很简单, 比如有两块 U 盘, 那么我可以一起都插入, 但是只需要加载一个模块, `usb-storage`。没听说过插入两块 U 盘就得加载两次驱动程序的, 除非这两块 U 盘本身就得使用不同的驱动程序。也正是因为一个模块可以被多个设备共用, 才会有“模块计数”这个说法。

既然一个驱动可以支持多个设备, 那么当发现一个设备时, 如何知道哪个才是它的驱动呢? 这就是 `id_table` 的用处, 让每一个 `struct usb_driver` 准备一张表, 里面注明该驱动支持哪些设备, 这总可以了吧。如果这个设备属于这张表里的, 那么绑定, 如果不属于这张表里的, 那么不好意思, 您请便。

`id_table` 来自 `struct usb_driver` 中:

```
const struct usb_device_id *id_table;
```

它实际上是一个指针, 一个 `struct usb_device_id` 结构体的指针, 当然赋了值以后就是代表一个数组名了, 正如我们在定义 `struct usb_driver usb_storage_driver` 中所赋的值那样, `.id_table=storage_usb_ids`, 那好, 我们来看一下 `usb_device_id` 这究竟是怎样一个结构体。

`struct usb_device_id` 来自 `include/linux/mod_devicetable.h`:

```
98 struct usb_device_id {
99     /* which fields to match against? */
100     __u16      match_flags;
101
102     /* Used for product specific matches; range is inclusive */
103     __u16      idVendor;
104     __u16      idProduct;
105     __u16      bcdDevice_lo;
106     __u16      bcdDevice_hi;
107
108     /* Used for device class matches */
109     __u8       bDeviceClass;
110     __u8       bDeviceSubClass;
```

```

111     __u8          bDeviceProtocol;
112
113     /* Used for interface class matches */
114     __u8          bInterfaceClass;
115     __u8          bInterfaceSubClass;
116     __u8          bInterfaceProtocol;
117
118     /* not matched against */
119     kernel_ulong_t driver_info;
120 };

```

实际上这个结构体对每一个 USB 设备来说，就相当于它的身份证，记录了它的一些基本信息。通常我们的身份证上会记录我们的姓名、性别、出生年月、户口地址等，而 USB 设备也有它需要记录的信息，以区分它和别的 USB 设备。比如 Vendor-厂家、Product-产品，以及其他一些比如产品编号、产品的类别、遵循的协议，这些都会在 USB 的规范里边找到对应的成员。

于是我们知道，一个 usb\_driver 会把它的 id 表和每一个 USB 设备的实际情况进行比较。如果该设备的实际情况和这张表里的某一个 id 相同，准确地说，只有这许多特征都吻合，才能够把一个 USB 设备和这个 USB 驱动进行绑定，这些特征哪怕差一点也不行。

那么 USB 设备的实际情况是什么时候建立起来的？在介绍 probe 指针之前有必要先谈一谈另一个数据结构了，它就是 struct usb\_device。

## 11. 从协议中来，到协议中去（上）

在 struct usb\_driver 中，.probe 和 .disconnect 的原型如下：

```

836     int (*probe) (struct usb_interface *intf,
837                  const struct usb_device_id *id);
838
839     void (*disconnect) (struct usb_interface *intf);

```

我们来看其中的参数，struct usb\_device\_id 这个不用说了，刚才已经介绍过，那么 struct usb\_interface 从何而来？还是让我们先从 struct usb\_device 说起。

我们知道每一个设备对应一个 struct device 结构体变量，但是设备不可能是万能的，生命是多样性的，就像我们可以用“人”来统称全人类，但是分得细一点，又有男人和女人的区别。那么设备也一样，由于有各种各样的设备，于是又出来了更多的词汇（数据结构），比如针对 USB 设备，开发人员们设计了一个叫做 struct usb\_device 的结构体。它定义于 include/linux/usb.h:

```

336 struct usb_device {
337     int          devnum;          /* Address on USB bus */
338     char          devpath [16]; /* Use in messages: /port/port/... */
339     enum usb_device_state state; /* configured, not attached, etc */
340     enum usb_device_speed speed; /* high/full/low (or error) */

```

```

341
342     struct usb_tt      *tt;           /* low/full speed dev, highspeed hub */
343     int                 ttport;       /* device port on that tt hub */
344
345     unsigned int toggle[2];          /* one bit for each endpoint
346                                       * ([0] = IN, [1] = OUT) */
347
348     struct usb_device *parent;       /* our hub, unless we're the root */
349     struct usb_bus *bus;             /* Bus we're part of */
350     struct usb_host_endpoint ep0;
351
352     struct device dev;               /* Generic device interface */
353
354     struct usb_device_descriptor descriptor; /* Descriptor */
355     struct usb_host_config *config; /* All of the configs */
356
357     struct usb_host_config *actconfig; /* the active configuration */
358     struct usb_host_endpoint *ep_in[16];
359     struct usb_host_endpoint *ep_out[16];
360
361     char **rawdescriptors;          /* Raw descriptors for each config */
362
363     unsigned short bus_mA;          /* Current available from the bus */
364     u8 portnum;                     /* Parent port number (origin 1) */
365     u8 level;                       /* Number of USB hub ancestors */
366
367     unsigned discon_suspended:1;    /* Disconnected while suspended */
368     unsigned have_langid:1;         /* whether string_langid is valid */
369     int string_langid;              /* language ID for strings */
370
371     /* static strings from the device */
372     char *product;                  /* iProduct string, if present */
373     char *manufacturer;            /* iManufacturer string, if present */
374     char *serial;                   /* iSerialNumber string, if present */
375
376     struct list_head filelist;
377 #ifdef CONFIG_USB_DEVICE_CLASS
378     struct device *usb_classdev;
379 #endif
380 #ifdef CONFIG_USB_DEVICEFS
381     struct dentry *usbfs_dentry; /*usbfs dentry entry for the device*/
382 #endif
383     /*
384     * Child devices - these can be either new devices
385     * (if this is a hub device), or different instances
386     * of this same device.
387     *
388     * Each instance needs its own set of data structures.
389     */
390
391     int maxchild;                   /* Number of ports if hub */
392     struct usb_device *children[USB_MAXCHILDREN];
393
394     int pm_usage_cnt;               /* usage counter for autosuspend */
395     u32 quirks;                     /* quirks of the whole device */
396
397 #ifdef CONFIG_PM
398     struct delayed_work autosuspend; /* for delayed autosuspends */
399     struct mutex pm_mutex;          /* protects PM operations */

```

```

400
401     unsigned long last_busy;          /* time of last use */
402     int autosuspend_delay;           /* in jiffies */
403
404     unsigned auto_pm:1;               /* autosuspend/resume in progress */
405     unsigned do_remote_wakeup:1; /* remote wakeup should be enabled */
406     unsigned autosuspend_disabled:1; /* autosuspend and autoresume */
407     unsigned autoresume_disabled:1; /* disabled by the user */
408 #endif
409 };
410 #define to_usb_device(d) container_of(d, struct usb_device, dev)

```

看起来很复杂的一个数据结构，不过我们目前不需要去理解她的每一个成员，不过我们可以看到，其中有一个成员 `struct device dev`，这就是前面说的那个属于每个设备的 `struct device` 结构体变量。

实际上，U 盘驱动里边并不会直接去处理这个结构体，因为对于一个 U 盘来说，它就是对这么一个 `struct usb_device` 的变量，这个变量由 USB Core 负责申请和赋值。但是我们需要记住这个结构体变量，因为日后我们调用 USB Core 提供的函数时，会把这个变量作为参数传递上去。因为很简单，要和 USB Core 交流，总得让人家知道我们是谁吧。比如后来要调用的一个函数，`usb_buffer_alloc`，它就需要这个参数。

而对 U 盘设备驱动来说，比这个 `struct usb_device` 更重要的数据结构是，`struct usb_interface`。走到这一步，我们不得不去了解一些 USB 设备的规范了，或者专业一点说，USB 协议。因为我们至少要知道什么是 USB 接口（Interface）。

## 12. 从协议中来，到协议中去（中）

任何事物都有其要遵守的规矩。USB 设备要遵循的就是 USB 协议。不管是软件还是硬件，在设计开始，总是要参考 USB 协议。怎么设计硬件？如何编写软件？不看 USB 协议，谁也不可能凭空想象出来。

USB 协议规定了，每个 USB 设备都得有一些基本的元素，称为描述符。有四类描述符是任何一种 USB 设备都得有的，它们是，设备描述符（Device Descriptor）、配置描述符（Configuration Descriptor）、接口描述符（Interface Descriptor）、端点描述符（Endpoint Descriptor）。描述符里的东西是一个设备出厂时就被厂家给固化在设备中了。这种东西不管怎样也改变不了，比如我有一个 Intel 的 U 盘，里面的固有信息肯定是在 Intel 出厂时就被烙在里边了，厂家早已把它的一切，烙上 Intel 印。所以当我插入 U 盘，用 `cat /proc/scsi/scsi` 命令看一下的话，“Vendor”一项显示的肯定是 Intel。

关于这几种描述符，USB Core 在总线扫描时就会去读取，获得里面的信息，其中，设备描

述符描述的是整个设备。注意了，这个设备和咱们一直讲的设备驱动里的设备是不一样的。因为一个 USB 设备实际上指的是一种宏观上的概念，它可以是一种多功能的设备，改革开放之后，多功能的东西越来越多了，比如外企常见的多功能一体机，就是集打印机、复印机、扫描仪、传真机于一体的设备。当然，这不属于 USB 设备，但是 USB 设备当然也有这种情况，比如电台 DJ 可能会用到的，一个键盘，上边带一个扬声器，它们用两个 USB 接口接到 USB Hub 上去，而设备描述符描述的就是这整个设备的特点。

那么如何配置描述符呢？老实说，对我们了解 U 盘驱动真是没有什么意义，但是作为一个有责任的人，此刻，我必须为它多说几句，一个设备可以有一种或者几种配置。没见过具体的 USB 设备？那么好，手机见过吧，每个手机都会有多种配置，或者说“设定”，比如，我的这款，Nokia 6300。手机语言可以设定为 English、繁体中文或简体中文。一旦选择了其中一种，那么手机里面所显示的所有的信息都是该种语言/字体。还有最简单的例子，操作模式也有好几种，标准、无声、会议等。基本上如果我设为“会议”，那么就是只振动不发声；要是设为无声，那么就什么动静也不会有，只能凭感觉了。那么 USB 设备的配置也是如此，不同的 USB 设备当然有不同的配置了，或者说需要配置哪些东西也会不一样。好了，关于配置，就说这么多。

对于 USB 设备驱动程序编写者来说，更为关键的是下面的接口和端点。先说接口。第一句，一个接口对应一个 USB 设备驱动程序。没错，还举前边那个例子。一个 USB 设备，两种功能，一个键盘，上面带一个扬声器，两个接口，那肯定得要两个驱动程序，一个是键盘驱动程序，一个是音频流驱动程序。“道上”的兄弟喜欢把这样两个整合在一起的东西叫做一个设备，我们用接口来区分这两者。于是有了我们前面提到的那个数据结构，`struct usb_interface`，它定义于 `include/linux/usb.h`：

```
140 struct usb_interface {
141     /* array of alternate settings for this interface,
142      * stored in no particular order */
143     struct usb_host_interface *altsetting;
144
145     struct usb_host_interface *cur_altsetting; /* the currently
146                                                  * active alternate setting */
147     unsigned num_altsetting; /* number of alternate settings */
148
149     int minor; /* minor number this interface is
150                * bound to */
151     enum usb_interface_condition; /* state of binding */
152     unsigned is_active:1; /* the interface is not suspended */
153     unsigned needs_remote_wakeup:1; /*driver requires remote wakeup*/
154
155     struct device dev; /* interface specific device info */
156     struct device *usb_dev; /* pointer to the usb class's device, if any*/
157     int pm_usage_cnt; /* usage counter for autosuspend */
158 };
159 #define to_usb_interface(d) container_of(d, struct usb_interface, dev)
160 #define interface_to_usbdev(intf) \
161     container_of(intf->dev.parent, struct usb_device, dev)
```

这个结构体贯穿整个 U 盘驱动程序，所以虽然我们不用去深入了解，但是需要记住，整个

U 盘驱动程序在后面任何一处提到的 `struct usb_interface` 都是同一个变量，这个变量是在 USB Core 总线扫描时就申请好了的。我们只需要直接用就是了，比如前面说过的 `storage_probe(struct usb_interface *intf, const struct usb_device_id *id)`，`storage_disconnect(struct usb_interface *intf)` 这两个函数中的参数 `intf`。

而这里 130 行的宏 `interface_to_usbdev`，也会用得着的，顾名思义，就是从一个 `struct usb_interface` 转换成一个 `struct usb_device`，我们说过了，有些函数需要的参数就是 `struct usb_device`，而不是 `usb_interface`，所以这种转换是经常会用到的，而这个宏，USB Core 的设计者们也为我们准备好了，除了感激，我们还能说什么呢？

---

## 13. 从协议中来，到协议中去（下）

如果你是急性子，那这时候你一定很想开始看 `storage_probe` 函数了，因为整个 U 盘的工作就是从这里开始的。

前面我们已经了解了设备、配置和接口，还剩最后一个就是端点。USB 通信的最基本的形式就是通过端点，一个接口有一个或多个端点，而作为像 U 盘这样的存储设备，它至少有一个控制端点，两个批量端点。这些端点都是干什么用的？说来话长，真是一言难尽啊。

USB 协议中规定了，USB 设备有四种通信方式，分别是控制传输、中断传输、批量传输、等时传输。其中，等时传输显然是用于音频和视频一类的设备，这类设备期望能够有一个比较稳定的数据流，比如你在网上 QQ 视频聊天，肯定希望每分钟传输的图像/声音速率是比较稳定的。`usb-storage` 里面肯定不会用到等时传输。因为我们只管复制一个文件，管它第一秒和第二秒的传输有什么区别，只要几十秒内传完了就行了。

相比之下，等时传输是四种传输中最麻烦的，所以，U 盘用不着。不过我要说，中断传输也用不着，对于 U 盘来说，的确用不着，虽然说 USB Mass Storage 的协议中间有一个叫做 CBI 的传输协议，CBI 就是 Control/Bulk/Interrupt，即控制/批量/中断，这三种传输都会用到，但这种传输协议并不适用于 U 盘，U 盘使用的是叫做 Bulk-Only 的传输协议。使用这种协议的设备只有两种传输方式：批量传输和控制传输，控制传输是任何一种 USB 设备都必须支持的，它专门用于传输一些控制信息。比如我想查询一下关于这个接口的一些信息，那么就用控制传输。而批量传输，它就是 U 盘的主要工作了，读写数据，这种情况就得用批量传输。具体的传输我们后面再讲。

好了，知道了传输方式，就可以来认识端点了。和端点齐名的有一个叫做管道（Pipe）。端点就是通信的发送点或者接收点，要发送数据，那你只要把数据发送到正确的端点那里就可以了。



之所以 U 盘有两个批量端点，是因为端点也是有方向的，一个叫做 Bulk IN，一个叫做 Bulk OUT。从 USB 主机到设备称为 OUT，从设备到主机称为 IN。而管道实际上只是为了让我们能够找到端点，就相当于我们日常说的邮编地址，比如一个国家，为了通信，我们必须给各个地方取名，给各条大大小小的路取名，比如要从某偏僻的小县城出发，要到北京，那你的这个端点就是北京，而你得知道你来北京的路线，那这个从你们县城到北京的路线就算一条管道。有人好奇地问了，管道应该有两个端点吧，一个端点是北京，那另一个端点呢？答案是，这条管道有些特殊，我们只需要知道一端的目的地是北京，而另一端是哪里无所谓，因为不管你在哪里你都得到北京。

严格来说，管道的另一端应该是 USB 主机，USB 协议中也是这么规定的，协议中管道代表着一种能力。怎样一种能力呢？在主机和设备上的端点之间移动数据，听上去挺玄的。因为事实上，USB 里面所有的数据传输都是由主机发起的。一切都是以主机为核心，各种设备紧紧围绕在主机周围。所以 USB Core 里面很多函数就是为了让 USB 主机能够正确地完成数据传输或者说传输调度，就得告诉主机这个管道，换言之，它得知道自己这次调度的数据是发送给谁，或者从谁那里传输过来。不过有人又要问了，比如说要从 U 盘里读一个文件，那告诉 USB 主机某个端点能有用吗？那个文件又不是存放在一个端点里面，它不是存放在 U 盘里面吗？这个就只能在后面的代码里去知道了。

---

## 14. 梦开始的地方

对于整个 usb-storage 模块，usb\_stor\_init() 函数是它的开始，然而，对于 U 盘驱动程序来说，它真正驱使 U 盘工作却始于 storage\_probe() 函数。

两条平行线只要相交，就注定开始纠缠一生，不管中间是否短暂分离。USB Core 为设备找到了适合它的驱动程序，或者为驱动程序找到了它所支持的设备，但这只是表明双方的第一印象还可以，但彼此是否真的适合对方还需要进一步了解。而 U 盘驱动则会调用函数 storage\_probe() 去认识对方，它是一个什么样的设备？这里调用了四个函数：get\_device\_info，get\_protocol，get\_transport，get\_pipes。

整个 U 盘驱动这部大戏，由 storage\_probe 开始，由 storage\_disconnect 结束。其中，storage\_probe 这个函数占了相当大的篇幅。我们一段一段地来看。这两个函数都来自 drivers/usb/storage/usb.c 中：

```
945 /* Probe to see if we can drive a newly-connected USB device */
946 static int storage_probe(struct usb_interface *intf,
947                         const struct usb_device_id *id)
948 {
949     struct Scsi_Host *host;
```

```

950     struct us_data *us;
951     int result;
952     struct task_struct *th;
953
954     if (usb_usual_check_type(id, USB_US_TYPE_STOR))
955         return -ENXIO;
956
957     US_DEBUGP("USB Mass Storage device detected\n");
958
959     /*
960      * Ask the SCSI layer to allocate a host structure, with extra
961      * space at the end for our private us_data structure.
962      */
963     host = scsi_host_alloc(&usb_stor_host_template, sizeof(*us));
964     if (!host) {
965         printk(KERN_WARNING USB_STORAGE
966              "Unable to allocate the scsi host\n");
967         return -ENOMEM;
968     }
969
970     us = host_to_us(host);
971     me mset(us, 0, sizeof(struct us_data));

```

首先先“贴”出这么几行，两个参数不用多说了，`struct usb_interface` 和 `struct usb_device_id` 的这两个指针都是前面介绍过的，来自 USB Core 一层，我们整个故事里用到的就是这么一个，这两个指针的指向是定下来的。

950 行，最重要的一个数据结构终于出现了。整个 `usb-storage` 模块里面自己定义的数据结构不多，但是 `us_data` 算一个。这个数据结构是跟随我们的代码一直走下去的，如影随形，几乎处处都可以看见它的身影。先把它的代码“贴”出来，来自 `drivers/usb/storage/usb.h`：

```

102 /* we allocate one of these for every device that we remember */
103 struct us_data {
104     /* The device we're working with
105      * It's important to note:
106      * (o) you must hold dev_mutex to change pushb_dev
107      */
108     struct mutex          dev_mutex;          /* protect pushb_dev */
109     struct usb_device      *pushb_dev;        /* this usb_device */
110     struct usb_interface  *pushb_intf;        /* this interface */
111     struct us_unusual_dev  *unusual_dev;      /* device-filter entry */
112     unsigned long         flags;              /* from filter initially */
113     unsigned int          send_bulk_pipe;     /* cached pipe values */
114     unsigned int          recv_bulk_pipe;
115     unsigned int          send_ctrl_pipe;
116     unsigned int          recv_ctrl_pipe;
117     unsigned int          recv_intr_pipe;
118
119     /* information about the device */
120     char                  *transport_name;
121     char                  *protocol_name;
122     __le32                bcs_signature;
123     u8                    subclass;
124     u8                    protocol;
125     u8                    max_lun;

```

```

126
127     u8                ifnum;          /* interface number */
128     u8                ep_bInterval;   /* interrupt interval */
129
130     /* function pointers for this device */
131     trans_cmnd         transport;      /* transport function */
132     trans_reset        transport_reset; /* transport device reset */
133     proto_cmnd         proto_handler;  /* protocol handler */
134
135     /* SCSI interfaces */
136     struct scsi_cmnd   *srb;           /* current srb */
137     unsigned int       tag;            /* current dCBWTag */
138
139     /* control and bulk communications data */
140     struct urb         *current_urb;   /* USB requests */
141     struct usb_ctrlrequest *cr;        /* control requests */
142     struct usb_sg_request current_sg;  /* scatter-gather req. */
143     unsigned char      *iobuf;        /* I/O buffer */
144     unsigned char      *sensebuf;     /* sense data buffer */
145     dma_addr_t         cr_dma;        /* buffer DMA addresses */
146     dma_addr_t         iobuf_dma;
147
148     /* mutual exclusion and synchronization structures */
149     struct semaphore    sema;          /* to sleep thread on */
150     struct completion   notify;        /* thread begin/end */
151     wait_queue_head_t   delay_wait;    /* wait during scan, reset */
152
153     /* subdriver information */
154     void                *extra;        /* Any extra data */
155     extra_data_destructor extra_destructor; /* extra data destructor */
156 #ifdef CONFIG_PM
157     pm_hook             suspend_resume_hook;
158 #endif
159 };

```

不难发现，Linux 内核中每一个重要的数据结构都很复杂。总之，这个令人头疼的数据结构是每一个设备都有的。换句话说，我们会为每一个设备申请一个 `us_data`，因为这个结构里的东西我们之后一直会用得着的。950 行，`struct us_data *us`，于是，日后我们会非常频繁地看到 `us`。另，`us` 什么意思？`us`，即 `usb storage`。

963 行，关于 `usb_stor_host_template`，必须得认真看一下，因为这个变量我们以后还会多次碰到。`scsi_host_alloc` 就是 SCSI 子系统提供的函数，它的作用就是申请一个 SCSI Host 相应的数据结构。而它的第 1 个参数 `&usb_stor_host_template`，其实是一个 `struct scsi_host_template` 的结构体指针，之所以 USB Mass Storage 里面会涉及 SCSI 这一层，是因为我们事实上把一块 U 盘模拟成了一块 SCSI 设备。对于 SCSI 设备来说，要想正常工作，得有一个 SCSI 卡，或者说 SCSI Host。而按照 SCSI 这一层的规矩，要想申请一个 SCSI Host 的结构体，我们就得提供一个 `struct scsi_host_template` 结构体，这其实从名字可以看出，是一个模板，SCSI 那层把一切都封装好了，只要提交一个模板给它，它就能为你提供一个 `struct Scsi_Host` 结构体。关于这个 `usb_stor_host_template`，它的定义或者说初始化是在 `drivers/usb/storage/scsiglue.c` 中：

```

441 struct scsi_host_template usb_stor_host_template = {

```

```

442  /* basic userland interface stuff */
443  .name = "usb-storage",
444  .proc_name = "usb-storage",
445  .proc_info = proc_info,
446  .info = host_info,
447
448  /* command interface -- queued only */
449  .queuecommand = queuecommand,
450
451  /* error and abort handlers */
452  .eh_abort_handler = command_abort,
453  .eh_device_reset_handler = device_reset,
454  .eh_bus_reset_handler = bus_reset,
455
456  /* queue commands only, only one command per LUN */
457  .can_queue = 1,
458  .cmd_per_lun = 1,
459
460  /* unknown initiator id */
461  .this_id = -1,
462
463  .slave_alloc = slave_alloc,
464  .slave_configure = slave_configure,
465
466  /* lots of sg segments can be handled */
467  .sg_tablesize = SG_ALL,
468
469  /* limit the total size of a transfer to 120 KB */
470  .max_sectors = 240,
471
472  /* merge commands... this see ms to help performance, but
473   * periodically someone should test to see which setting is more
474   * optimal.
475   */
476  .use_clustering = 1,
477
478  /* emulated HBA */
479  .emulated = 1,
480
481  /* we do our own delay after a device or bus reset */
482  .skip_settle_delay = 1,
483
484  /* sysfs device attributes */
485  .sdev_attrs = sysfs_device_attr_list,
486
487  /* module management */
488  .module = THIS_MODULE
489 };

```

按照 SCSI 层的规矩，要想申请一个 SCSI Host，并且让它工作，我们需要调用三个函数，第 1 个是 `scsi_host_alloc()`，第 2 个是 `scsi_add_host()`，第 3 个是 `scsi_scan_host()`。这三个函数我们都会在接下来的代码里面看到。

`scsi_host_alloc()` 一调用，就是给 `struct Scsi_Host` 结构申请了空间，而只有调用了 `scsi_add_host()` 之后，SCSI 核心层才知道有这个 Host 的存在，然后只有 `scsi_scan_host()` 被调用

了之后，真正的设备才被发现。这些函数的含义和它们的名字吻合得很好，不是吗？

最后需要指出的是，`scsi_host_alloc()`需要两个参数，第 1 个参数是 `struct scsi_host_template` 的指针，咱们当然给了它 `&usb_stor_host_template`，而第 2 个参数实际上是被称为驱动自己的数据，咱们传递的是 `sizeof(*us)`。SCSI 层非常友好地给咱们提供了一个接口，在 `struct Scsi_Host` 结构体被设计时就专门准备了一个 `unsigned long hostdata[0]` 来给别的设备驱动使用，这个 `hostdata` 的大小是可以由咱们来定的，把 `sizeof(*us)` 传递给 `scsi_host_alloc()` 就意味着给 `us` 申请了内存。而今后如果我们需要从 `us` 得到相应的 SCSI Host 就可以使用内联函数 `us_to_host()`；而反过来要想从 SCSI Host 得到相应的 `us` 则可以使用内联函数 `host_to_us()`，这两个明显是一对，都定义于 `drivers/usb/storage/usb.h`：

```
161 /* Convert between us_data and the corresponding Scsi_Host */
162 static inline struct Scsi_Host *us_to_host(struct us_data *us) {
163     return container_of((void *) us, struct Scsi_Host, hostdata);
164 }
165 static inline struct us_data *host_to_us(struct Scsi_Host *host) {
166     return (struct us_data *) host->hostdata;
167 }
```

总之咱们这么一折腾，就让 USB 驱动和 SCSI Host 联系起来。从此以后这个 U 盘既要扮演 U 盘的角色又要扮演 SCSI 设备的角色。

957 行，`US_DEBUGP` 是一个宏，来自 `drivers/usb/storage/debug.h`，接下来很多代码中我们也会看到这个宏，它无非就是打印一些调试信息。`debug.h` 中有这么一段：

```
51 #ifdef CONFIG_USB_STORAGE_DEBUG
52 void usb_stor_show_command(struct scsi_cmnd *srb);
53 void usb_stor_show_sense( unsigned char key,
54                          unsigned char asc, unsigned char ascq );
55 #define US_DEBUGP(x...) printk( KERN_DEBUG USB_STORAGE x )
56 #define US_DEBUGPX(x...) printk( x )
57 #define US_DEBUG(x) x
58 #else
59 #define US_DEBUGP(x...)
60 #define US_DEBUGPX(x...)
61 #define US_DEBUG(x)
62 #endif
```

这里一共定义了几个宏：`US_DEBUGP`，`US_DEBUGPX`，`US_DEBUG`，差别不大，只是形式上略有不同罢了。

需要注意的是，这些调试信息只有当我们打开了编译选项时 `CONFIG_USB_STORAGE_DEBUG` 才有意义的，如果这个选项为 0，那么这几个宏就什么也不干，因为它们被赋为空了。关于 `US_DEBUG` 系列的这几个宏，就讲到这了。

954 行，`usb_usual_check_type()` 干什么用的？这个函数来自 `drivers/usb/storage/libusual.c` 中：

```
98 int usb_usual_check_type(const struct usb_device_id *id, int caller_type)
99 {
```

```

100     int id_type = USB_US_TYPE(id->driver_info);
101
102     if (caller_type <= 0 || caller_type >= 3)
103         return -EINVAL;
104
105     /* Drivers grab fixed assignment devices */
106     if (id_type == caller_type)
107         return 0;
108     /* Drivers grab devices biased to them */
109     if (id_type==USB_US_TYPE_NONE && caller_type==atomic_read(&usu_bias))
110         return 0;
111     return -ENODEV;
112 }

```

这个函数就是保证现在认领的这个设备属于 `usb-storage` 所支持的设备，而不是另一个叫做 `ub` 的驱动所支持的设备，如果你足够细心可能会注意到，`drivers/block` 目录下面竟然也有一段与 `usb` 相关的驱动代码，它就是 `drivers/block/ub.c`。实际上 `ub` 是一个简化版的 `usb-storage` 和 `sd` 两个模块的结合体，它的功能比较弱，但是要更稳定。如果感兴趣的话可以去读一下 `ub` 这个模块的代码。

## 15. 设备花名册

`storage_probe` 这个函数挺有意思，总长度不足 100 行，但是干了许多事情，这就像足球场上的后腰，比如切尔西的马克莱莱，在场上并不起眼，但是却为整个团队作出了卓越的贡献。

我们继续看 `storage_probe` 的代码：

```

972     mutex_init(&(us->dev_mutex));
973     init_MUTEX_LOCKED(&(us->sema));
974     init_completion(&(us->notify));
975     init_waitqueue_head(&us->delay_wait);
976
977     /* Associate the us_data structure with the USB device */
978     result = associate_dev(us, intf);
979     if (result)
980         goto BadDevice;
981
982     /*
983     * Get the unusual_devs entries and the descriptors
984     *
985     * id_index is calculated in the declaration to be the index number
986     * of the match from the usb_device_id table, so we can find the
987     * corresponding entry in the private table.
988     */
989     result = get_device_info(us, id);
990     if (result)
991         goto BadDevice;
992

```

`storage_probe` 函数之所以短小，是因为它调用了大量的函数。所以，看起来短短一段代码，实际上却要花费读代码的人好几个小时。

## 16. 冰冻三尺非一日之寒

罗马不是一天建成的。在让 U 盘工作之前，其实我们的驱动做了很多准备工作。

我们继续跟着感觉走，`storage_probe()`，972 行至 975 行，一系列以 `init_*` 命名的函数在此刻被调用，这里涉及了一些锁机制、等待机制，不过只是初始化，暂且不理睬，到后面用到时再细说，不过请记住，这几行每一行都是有用的。后面自然会用得着。

此时，我们先往下走，978 行的 `associate_dev()` 和 989 行的 `get_device_info()`，这两个函数是我们目前需要看的。

先看 `associate_dev()` 函数，定义于 `drivers/usb/storage/usb.c` 中。

```
438 /* Associate our private data with the USB device */
439 static int associate_dev(struct us_data *us, struct usb_interface *intf)
440 {
441     US_DEBUGP("-- %s\n", __FUNCTION__);
442
443     /* Fill in the device-related fields */
444     us->pusb_dev = interface_to_usbdev(intf);
445     us->pusb_intf = intf;
446     us->ifnum = intf->cur_altsetting->desc.bInterfaceNumber;
447     US_DEBUGP("Vendor: 0x%04x, Product: 0x%04x, Revision: 0x%04x\n",
448               le16_to_cpu(us->pusb_dev->descriptor.idVendor),
449               le16_to_cpu(us->pusb_dev->descriptor.idProduct),
450               le16_to_cpu(us->pusb_dev->descriptor.bcdDevice));
451     US_DEBUGP("Interface Subclass: 0x%02x, Protocol: 0x%02x\n",
452               intf->cur_altsetting->desc.bInterfaceSubClass,
453               intf->cur_altsetting->desc.bInterfaceProtocol);
454
455     /* Store our private data in the interface */
456     usb_set_intfdata(intf, us);
457
458     /* Allocate the device-related DMA-mapped buffers */
459     us->cr = usb_buffer_alloc(us->pusb_dev, sizeof(*us->cr),
460                              GFP_KERNEL, &us->cr_dma);
461     if (!us->cr) {
462         US_DEBUGP("usb_ctrlrequest allocation failed\n");
463         return -ENOMEM;
464     }
465
466     us->iobuf = usb_buffer_alloc(us->pusb_dev, US_IOBUF_SIZE,
467                                 GFP_KERNEL, &us->iobuf_dma);
468     if (!us->iobuf) {
469         US_DEBUGP("I/O buffer allocation failed\n");
```

```

470         return -ENOMEM;
471     }
472
473     us->sensebuf = kmalloc(US_SENSE_SIZE, GFP_KERNEL);
474     if (!us->sensebuf) {
475         US_DEBUGP("Sense buffer allocation failed\n");
476         return -ENOMEM;
477     }
478     return 0;
479 }

```

我们首先来关注函数 `associate_dev` 的参数, `struct us_data *us`, 传递给它的是 `us`, 这个不用多说了吧, 此前刚刚为它申请了内存, 并且初始化各成员为 0。这个 `us` 将一直陪伴我们走下去, 直到故事结束。所以其重要性不言而喻。`struct usb_interface *intf`, `storage_probe()` 函数传进来的两个参数之一。

总之, 此处郑重申明一次, `struct us_data` 的结构体指针 `us`、`struct usb_interface` 结构体的指针 `intf`, 以及 `struct usb_device` 结构体和 `struct usb_device_id` 结构体在整个 U 盘驱动的故事中是唯一的, 每次提到都是那个。而以后我们会遇上的几个重要的数据结构: `struct urb urb`, `struct scsi_cmnd srb` 也非常重要, 但是它们并不唯一, 也许每次遇上都不一样, 就像演戏一样。前边这几个数据结构的变量就像那些主角, 而之后遇见的 `urb`、`srb`, 虽然频繁露面, 但是只是群众演员, 只不过这次是路人甲, 下次是路人乙。所以, 以后我们将只说 `us`, 不再说 `struct us_data *us`, `struct usb_interface *intf` 也将只用 `intf` 来代替。

`us` 之所以重要, 是因为接下来很多函数都要用到它, 以及它的各个成员。实际上目前这个函数 `associate_dev` 所做的事情就是为 `us` 的各个成员赋值, 毕竟此刻 `us` 和我们之前提到的那些函数 `struct usb_device`, `struct usb_interface` 还没有一点关系。因而, 这个函数, 以及这之后的好几个函数都是为了给 `us` 的各成员赋上适当的值, 之所以如此兴师动众去为它赋值, 主要是因为后面要利用它。正所谓天下没有免费的午餐。

441 行, 本来无须多讲, 因为只是一个 `debug` 语句, 不过提一下 `__FUNCTION__` 这个宏, GCC 2.95 以后的版本支持这个宏在编译时会被转换为函数名 (字符串), 这里自然就是 “`associate_dev`” 这个字符串, 于是函数执行到这里就会打印一句话告诉世人我们执行到这个函数了, 这种做法显然会有利于调试程序。不过这个东西实际上不是宏, 因为预处理器对它一无所知。它的心只有编译器才懂。

444 行, `pusb_dev`, 意思是 `point of usb device`, `struct us_data` 中的一个成员。按照我们刚才约定的规矩, 此刻我将说它是 `us` 的一个成员, `us->pusb_dev = interface_to_usbdev(intf)`, `interface_to_usbdev` 我们前面已经讲过, 其含义是把一个 `struct interface` 结构体的指针转换成一个 `struct usb_device` 的结构体指针。前面说过, `struct usb_device` 对我们没有什么用, 但是 USB Core 层的一些函数要求使用这个参数, 所以我们不得已而为之, 正所谓人在江湖身不由己。

445 行, 把 `intf` 赋给 `us` 的 `pusb_intf`。



446 行, `us` 的 `ifnum`, 先看 `intf` 的 `cur_altsetting`, 这个容易令外行混淆。USB 设备有一个配置, 这个我们前面讲协议时讲了, 而这里又有一个 `setting` (设置)。乍一看有些奇怪, 这两个词不是一回事吗? 还是拿我们最熟悉的手机来打比方, 配置不说了, 只说设置。一个手机可能各种配置都确定了, 如是振动还是铃声, 各种功能都确定了, 但是声音的大小还可以改变, 通常手机的音量是一格一格地变动, 大概也就五六格, 那么这个可以算一个设置。这里 `cur_altsetting` 就是表示当前的这个设置。`cur_altsetting` 是一个 `struct usb_host_interface` 的指针, 这个结构体定义于 `include/linux/usb.h`:

```
69 /* host-side wrapper for one interface setting's parsed descriptors */
70 struct usb_host_interface {
71     struct usb_interface_descriptor desc;
72
73     /* array of desc.bNumEndpoint endpoints associated with this
74      * interface setting. these will be in no particular order.
75      */
76     struct usb_host_endpoint *endpoint;
77
78     char *string;          /* iInterface string, if present */
79     unsigned char *extra; /* Extra descriptors */
80     int extralen;
81 };
```

它的成员 `desc` 是一个 `struct usb_interface_descriptor` 结构体变量, 这个结构体的定义是和 USB 协议直接对应的, 定义于 `include/linux/usb/ch9.h`。(这里取名为 `ch9` 是因为这个文件中很多东西对应于 USB spec 2.0 中的第 9 章, chapter 9。)

```
294 /* USB_DT_INTERFACE: Interface descriptor */
295 struct usb_interface_descriptor {
296     __u8 bLength;
297     __u8 bDescriptorType;
298
299     __u8 bInterfaceNumber;
300     __u8 bAlternateSetting;
301     __u8 bNumEndpoints;
302     __u8 bInterfaceClass;
303     __u8 bInterfaceSubClass;
304     __u8 bInterfaceProtocol;
305     __u8 iInterface;
306 } __attribute__((packed));
```

而其中我们这里提到的是 `bInterfaceNumber`, 一个设备可以有多个接口, 于是每一个接口当然就得用一个编号了, 要不然怎么区分啊? 所有这些描述符里的东西都是出厂时就固化在设备中的, 而我们这里之所以可以用 `bInterfaceNumber` 来赋值, 是因为 USB Core 在为设备初始化时就已经做足了功课, 否则的话, 我们真是寸步难行。

总之, `us->ifnum` 就是这样, 最终就是等于咱们眼下这个接口的编号。

447 行到 453 行就是两句调试语句, 打印更多的描述符信息, 包括设备描述符和接口描述符。

447 行，`usb_set_intfdata()`，这其实是一个内联函数，就一行代码，也是定义于 `include/linux/usb.h` 中：

```
168 static inline void usb_set_intfdata (struct usb_interface *intf, void *data)
169 {
170     dev_set_drvdata(&intf->dev, data);
171 }
```

有趣的是，`dev_set_drvdata` 这个函数也是内联函数，也只有一行代码，它定义于 `include/linux/device.h` 中：

```
491 static inline void
492 dev_set_drvdata (struct device *dev, void *data)
493 {
494     dev->driver_data = data;
495 }
```

所以，结合来看，最终做的事情就是让 `&intf->dev->driver_data=data`，即 `&intf->dev->driver_data=us`。

再往下走，就是申请内存了，`us->cr` 和 `us->iobuf` 都是指针，这里让它们指向两段内存空间，下面会用到。需要注意的是，`usb_buffer_alloc()`这个函数是 USB Core 提供的，我们只管调用即可。从名字上就能知道它是用来申请内存的，第 1 个参数就是 `struct usb_device` 结构体的指针，所以我们要传递一个 `pusb_dev`。第 3 个参数，`GFP_KERNEL` 是一个内存申请的 `flag`，通常内存申请都用这个 `flag`，除非是中断上下文，不能睡眠，那就得用 `GPF_ATOMIC`，这里没那么多要求。第 2 个参数申请的 `buffer` 的大小，对于 `cr`，传递的是 `sizeof(*us->cr)`，而对于 `iobuf`，传递的是 `US_IOBUF_SIZE`，这是一个宏，大小是 64，是我们自己定义的，来自 `drivers/usb/storage/usb.h`：

```
90 #define US_IOBUF_SIZE      64 /* Size of the DMA-mapped I/O buffer */
91 #define US_SENSE_SIZE      18 /* Size of the autosense data buffer */
```

而 `usb_buffer_alloc()` 的第 4 个参数很有意思，第一次我们传递的是 `&us->cr_dma`，第二次传递的是 `&us->iobuf_dma`，这涉及 DMA 传输。这两个参数此前我们都没有赋过值，相反它们是在这个函数调用之后被赋上值的。`cr_dma` 和 `iobuf_dma` 都是 `dma_addr_t` 类型的变量，这个数据类型是 Linux 内核中专门为 DMA 传输而准备的。为了支持 DMA 传输，`usb_buffer_alloc` 不仅仅是申请了地址，并且建立了 DMA 映射，`cr_dma` 和 `iobuf_dma` 就是记录着 `cr` 和 `iobuf` 的 `dma` 地址。关于什么是 `cr`，关于这些 DMA 地址究竟有什么用，我们稍后就会遇到，那时候再讲也不迟。现在需要知道的就是 `usb_buffer_alloc` 申请的空间分别返回给了 `cr` 和 `iobuf`。顺便提一下，用 `usb_buffer_alloc` 申请的内存空间需要用它的搭档 `usb_buffer_free()` 来释放。

461 行和 468 行，每一次申请完内存就要检查成功与否，这是惯例。驱动程序能否驱动设备，关键就是看能否申请到内存空间，任何一处内存空间申请失败，整个驱动程序就没法正常工作。

此外还看到 473 行，我们还为 `us->sensebuf` 申请了内存，关于 `sense buffer`，等到讲 SCSI 命令数据传输时再来看，现在还不需要了解。

## 17. 冬天来了，春天还会远吗？（一）

在整个 `usb-storage` 模块的代码中，其最灵魂的部分在一个叫做 `usb_stor_control_thread()` 的函数中，而那也自然是我们整个故事的高潮。这个函数的调用有一些特殊，是从 `usb_stor_acquire_resources()` 函数进入的，而后者我们即将遇到，它在整部戏中只出现过一次，即 `storage_probe` 中，行号为 1005 的地方。

然而在此之前，有四个函数挡在我们面前，它们就是 `get_device_info`，`get_transport`，`get_protocol`，`get_pipes`。如我前面所说，两个人要走到一起，首先要了解彼此，这四个函数就是让驱动去认识设备的。这一点我们从名字上也能看出来。驱动需要知道设备的名字，所以有了 `get_device_info`，驱动需要知道设备是哪一种类型，写代码的人把这些工作分配给了 `get_transport`，`get_protocol` 和 `get_pipes`。

实际上，这四个函数，加上之前刚说过的 `associate_dev()` 函数，是整个故事中最平淡最枯燥的部分，第一次读这部分代码总让人困惑，怎么没看见一点 USB 数据通信？完全没有看到 USB 主机和 USB 设备是如何在交流的，这是 USB 吗？这几个函数应该说是给后面做铺垫，红花总要有绿叶配，没有这段代码的铺垫，到了后面 USB 设备恐怕也无法正常工作吧。不过，一个好消息是，这几个函数我们只会遇见这一次，它们在整个故事中就这么一次露脸的机会，像我们每个人的青春，只有一次，无法回头。所以，让我们享受这段平淡无奇的代码吧。

`get_device_info`，这个函数定义于 `drivers/usb/storage/usb.c` 中：

```
488 /* Get the unusual_devs entries and the string descriptors */
489 static int get_device_info(struct us_data *us, const struct usb_device_id
*id)
490 {
491     struct usb_device *dev = us->pusb_dev;
492     struct usb_interface_descriptor *idesc =
493         &us->pusb_intf->cur_altsetting->desc;
494     struct us_unusual_dev *unusual_dev = find_unusual(id);
495
496     /* Store the entries */
497     us->unusual_dev = unusual_dev;
498     us->subclass = (unusual_dev->useProtocol == US_SC_DEVICE) ?
499         idesc->bInterfaceSubClass :
500         unusual_dev->useProtocol;
501     us->protocol = (unusual_dev->useTransport == US_PR_DEVICE) ?
502         idesc->bInterfaceProtocol :
503         unusual_dev->useTransport;
504     us->flags = USB_US_ORIG_FLAGS(id->driver_info);
```

```

505
506     if (us->flags & US_FL_IGNORE_DEVICE) {
507         printk(KERN_INFO USB_STORAGE "device ignored\n");
508         return -ENODEV;
509     }
510
511     /*
512     * This flag is only needed when we're in high-speed, so let's
513     * disable it if we're in full-speed
514     */
515     if (dev->speed != USB_SPEED_HIGH)
516         us->flags &= ~US_FL_GO_SLOW;
517
518     /* Log a message if a non-generic unusual_dev entry contains an
519     * unnecessary subclass or protocol override. This may stimulate
520     * reports from users that will help us remove unneeded entries
521     * from the unusual_devs.h table.
522     */
523     if (id->idVendor || id->idProduct) {
524         static const char * msgs[3] = {
525             "an unneeded SubClass entry",
526             "an unneeded Protocol entry",
527             "unneeded SubClass and Protocol entries"};
528         struct usb_device_descriptor *ddesc = &dev->descriptor;
529         int msg = -1;
530
531         if (unusual_dev->useProtocol != US_SC_DEVICE &&
532             us->subclass == idesc->bInterfaceSubClass)
533             msg += 1;
534         if (unusual_dev->useTransport != US_PR_DEVICE &&
535             us->protocol == idesc->bInterfaceProtocol)
536             msg += 2;
537         if (msg >= 0 && !(us->flags & US_FL_NEED_OVERRIDE))
538             printk(KERN_NOTICE USB_STORAGE "This device "
539                 " (%04x,%04x,%04x S %02x P %02x)"
540                 " has %s in unusual_devs.h (kernel"
541                 " %s)\n"
542                 "   Please send a copy of this message to "
543                 "<linux-usb-devel@lists.sourceforge.net>\n",
544                 le16_to_cpu(ddesc->idVendor),
545                 le16_to_cpu(ddesc->idProduct),
546                 le16_to_cpu(ddesc->bcdDevice),
547                 idesc->bInterfaceSubClass,
548                 idesc->bInterfaceProtocol,
549                 msgs[ msg],
550                 utsname()->release);
551     }
552
553     return 0;
554 }

```

492 行, struct usb\_interface\_descriptor \*idesc, 这个也无需再说, 在之前的 associate\_dev 函数中已经介绍过这个结构体, 而且整个故事就是针对一个接口的, 一个接口就对应一个接口描述符。

494 行, struct us\_unusual\_dev, 这个结构体是第一次出现, 它定义于 drivers/usb/storage/usb.h

中：

```
61 struct us_unusual_dev {
62     const char* vendorName;
63     const char* productName;
64     __u8 useProtocol;
65     __u8 useTransport;
66     int (*initFunction)(struct us_data *);
67 };
```

而 “=” 右边的 `find_unusual()` 函数定义于 `drivers/usb/storage/usb.c` 中：

```
482 static struct us_unusual_dev *find_unusual(const struct usb_device_id *id)
483 {
484     const int id_index = id - storage_usb_ids;
485     return &us_unusual_dev_list[id_index];
486 }
```

`us_unusual_dev_list` 是一个数组，定义于 `drivers/usb/storage/usb.c`：

```
178 static struct us_unusual_dev us_unusual_dev_list[] = {
179 #     include "unusual_devs.h"
180 #     undef UNUSUAL_DEV
181 #     undef USUAL_DEV
182
183     /* Terminating entry */
184     { NULL }
185 };
```

Linux 代码中有很多奇怪的地方，可是像 `us_unusual_dev_list` 这个数组这么奇怪还真没见过。为了了解这个数组以及 `find_unusual()` 函数，我们先来看一看这个 `storage_usb_ids`。它不是别人，正是我们曾经赋给 `usb_storage_driver` 的成员 `id_table` 的值。忘记了 `id_table` 的可以回去看。它实际上就是一张表格，告诉全世界 driver 支持怎样的一些设备。`storage_usb_ids` 同样来自 `drivers/usb/storage/usb.c` 中：

```
138 static struct usb_device_id storage_usb_ids [] = {
139
140 #     include "unusual_devs.h"
141 #undef UNUSUAL_DEV
142 #undef USUAL_DEV
143     /* Terminating entry */
144     { }
145 };
```

这么一看，`us_unusual_dev_list` 和 `storage_usb_ids` 俨然是双胞胎啊！唯一的区别只是前者多了一个 `NULL`。这里最莫名其妙的就是包含了一个文件，于是让我们先来查看这个 `unusual_devs.h` 文件到底是干什么的？先看一下这个文件最下面的一些行：

```
1476 /* Control/Bulk transport for all SubClass values */
1477 USUAL_DEV(US_SC_RBC, US_PR_CB, USB_US_TYPE_STOR),
1478 USUAL_DEV(US_SC_8020, US_PR_CB, USB_US_TYPE_STOR),
1479 USUAL_DEV(US_SC_QIC, US_PR_CB, USB_US_TYPE_STOR),
1480 USUAL_DEV(US_SC_UFI, US_PR_CB, USB_US_TYPE_STOR),
1481 USUAL_DEV(US_SC_8070, US_PR_CB, USB_US_TYPE_STOR),
```

```

1482 USUAL_DEV(US_SC_SCSI, US_PR_CB, USB_US_TYPE_STOR),
1483
1484 /* Control/Bulk/Interrupt transport for all SubClass values */
1485 USUAL_DEV(US_SC_RBC, US_PR_CBI, USB_US_TYPE_STOR),
1486 USUAL_DEV(US_SC_8020, US_PR_CBI, USB_US_TYPE_STOR),
1487 USUAL_DEV(US_SC_QIC, US_PR_CBI, USB_US_TYPE_STOR),
1488 USUAL_DEV(US_SC_UFI, US_PR_CBI, USB_US_TYPE_STOR),
1489 USUAL_DEV(US_SC_8070, US_PR_CBI, USB_US_TYPE_STOR),
1490 USUAL_DEV(US_SC_SCSI, US_PR_CBI, USB_US_TYPE_STOR),
1491
1492 /* Bulk-only transport for all SubClass values */
1493 USUAL_DEV(US_SC_RBC, US_PR_BULK, USB_US_TYPE_STOR),
1494 USUAL_DEV(US_SC_8020, US_PR_BULK, USB_US_TYPE_STOR),
1495 USUAL_DEV(US_SC_QIC, US_PR_BULK, USB_US_TYPE_STOR),
1496 USUAL_DEV(US_SC_UFI, US_PR_BULK, USB_US_TYPE_STOR),
1497 USUAL_DEV(US_SC_8070, US_PR_BULK, USB_US_TYPE_STOR),
1498 USUAL_DEV(US_SC_SCSI, US_PR_BULK, 0),

```

USUAL\_DEV 以及 UNUSUAL\_DEV 均定义于 drivers/usb/storage/usb.c 中:

```

128 #define UNUSUAL_DEV(id_vendor, id_product, bcdDeviceMin, bcdDeviceMax, \
129                     vendorName, productName, useProtocol, useTransport, \
130                     initFunction, flags) \
131 { USB_DEVICE_VER(id_vendor, id_product, bcdDeviceMin, bcdDeviceMax), \
132   .driver_info = (flags) | (USB_US_TYPE_STOR << 24) }
133
134 #define USUAL_DEV(useProto, useTrans, useType) \
135 { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, useProto, useTrans), \
136   .driver_info = (USB_US_TYPE_STOR << 24) }

```

注意到我们看的是 struct usb\_device\_id 结构体的数组，其中每一项必然是一个 struct usb\_device\_id 的结构体变量。我们先来看 USB\_DEVICE\_VER 和 USB\_INTERFACE\_INFO，很显然这两个都是宏，来自 include/linux/usb.h:

```

715 /**
716  * USB_DEVICE_VER - macro used to describe a specific usb device with a
717  *                   version range
718  * @vend: the 16 bit USB Vendor ID
719  * @prod: the 16 bit USB Product ID
720  * @lo: the bcdDevice_lo value
721  * @hi: the bcdDevice_hi value
722  *
723  * This macro is used to create a struct usb_device_id that matches a
724  * specific device, with a version range.
725  */
726 #define USB_DEVICE_VER(vend, prod, lo, hi) \
727     .match_flags = USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, \
728     .idVendor = (vend), .idProduct = (prod), \
729     .bcdDevice_lo = (lo), .bcdDevice_hi = (hi)
744 /**
745  * USB_INTERFACE_INFO - macro used to describe a class of usb interfaces
746  * @cl: bInterfaceClass value
747  * @sc: bInterfaceSubClass value
748  * @pr: bInterfaceProtocol value
749  *
750  * This macro is used to create a struct usb_device_id that matches a
751  * specific class of interfaces.

```

```

752 */
753 #define USB_INTERFACE_INFO(cl,sc,pr) \
754     .match_flags = USB_DEVICE_ID_MATCH_INT_INFO, .bInterfaceClass = (cl),\
755     .bInterfaceSubClass = (sc), .bInterfaceProtocol = (pr)
756

```

每一个 `USB_DEVICE_VER` 或者 `USB_INTERFACE_INFO` 就是构造一个 `struct usb_device_id` 的结构体变量，回顾一下 `struct usb_device_id` 的定义，这里实际上就是为其中的四个元素赋了值，它们是 `match_flags`、`bInterfaceClass`、`bInterfaceSubClass` 和 `bInterfaceProtocol`。

这里不得不说的是，这个世界上有许许多多的 USB 设备，它们各有各的特点，为了区分它们，USB 规范或者说 USB 协议，把 USB 设备分成了很多类，然而每个类又分成子类。这很好理解，我们的大学也是如此，先是分成很多个学院，然后每个学院又被分为很多个系，比如信息学院，下面分了电子工程系、微电子系、计算机系、通信工程系，然后可能每个系下边又分了各个专业。USB 协议也是，首先每个接口属于一个 Class，（为什么是把接口分类，而不把设备分类？前面讲过了，在 USB 设备驱动中，不用再提设备，因为每个设备驱动对应的是一种接口，而不是一种设备），然后 Class 下面分了 SubClass，接着 SubClass 下面又按各种设备所遵循的不同的通信协议继续细分。

USB 协议中为每一种 Class、每一种 SubClass 和每一种 Protocol 定义一个数值，比如 Mass Storage 的 Class 就是 `0x08`，而这里 `USB_CLASS_MASS_STORAGE` 这个宏在 `include/linux/usb/ch9.h` 中定义，其值正是 8。

我们拿第 1477 行来举例。

```

1477 USUAL_DEV(US_SC_RBC, US_PR_CB, USB_US_TYPE_STOR),

```

把这个宏展开，就是说定义了这么一个 `usb_device_id` 结构体变量，其 `match_flags=USB_DEVICE_ID_MATCH_INT_INFO`，而 `bInterfaceClass=USB_CLASS_MASS_STORAGE`，`bInterfaceSubClass=US_SC_RBC`，以及 `bInterfaceProtocol=US_PR_CB`。

`USB_CLASS_MASS_STORAGE` 就不用再说了，这个驱动程序所支持的每一种设备都是属于这个类，或者说这个 Class。但是这个 Class 里面包含不同的 SubClass，比如 SubClass 02 为 CD-ROM 设备，04 为软盘驱动器，06 为通用 SCSI 类设备。而通信协议则主要有 CBI 协议和 Bulk-Only 协议。

像 `US_SC_RBC` 这些关于 SubClass 的宏的定义是在文件 `include/linux/usb_usual.h` 中：

```

74 #define US_SC_RBC      0x01      /* Typically, flash devices */
75 #define US_SC_8020     0x02      /* CD-ROM */
76 #define US_SC_QIC      0x03      /* QIC-157 Tapes */
77 #define US_SC_UFI      0x04      /* Floppy */
78 #define US_SC_8070     0x05      /* Removable media */
79 #define US_SC_SCSI     0x06      /* Transparent */

```

```

80 #define US_SC_ISD200    0x07          /* ISD200 ATA */
81 #define US_SC_MIN      US_SC_RBC
82 #define US_SC_MAX      US_SC_ISD200
83
84 #define US_SC_DEVICE    0xff          /* Use device's value */

```

而像 `US_PR_CB` 这些关于传输协议的宏也定义于同一个文件中：

```

88 #define US_PR_CBI      0x00          /* Control/Bulk/Interrupt */
89 #define US_PR_CB       0x01          /* Control/Bulk w/o interrupt */
90 #define US_PR_BULK     0x50          /* bulk only */
91 #ifdef CONFIG_USB_STORAGE_USBAT
92 #define US_PR_USBAT    0x80          /* SCM-ATAPI bridge */
93 #endif
94 #ifdef CONFIG_USB_STORAGE_SDDR09
95 #define US_PR_EUSB_SDDR09 0x81      /* SCM-SCSI bridge for SDDR-09 */
96 #endif
97 #ifdef CONFIG_USB_STORAGE_SDDR55
98 #define US_PR_SDDR55   0x82          /* SDDR-55 (made up) */
99 #endif
100 #define US_PR_DPCM_USB 0xf0          /* Combination CB/SDDR09 */
101 #ifdef CONFIG_USB_STORAGE_FREECOM
102 #define US_PR_FREECOM  0xf1          /* Freecom */
103 #endif
104 #ifdef CONFIG_USB_STORAGE_DATAFAB
105 #define US_PR_DATAFAB  0xf2          /* Datafab chipsets */
106 #endif
107 #ifdef CONFIG_USB_STORAGE_JUMPSHOT
108 #define US_PR_JUMPSHOT 0xf3          /* Lexar Jumpshot */
109 #endif
110 #ifdef CONFIG_USB_STORAGE_ALAUDA
111 #define US_PR_ALAUDA   0xf4          /* Alauda chipsets */
112 #endif
113 #ifdef CONFIG_USB_STORAGE_KARMA
114 #define US_PR_KARMA    0xf5          /* Rio Karma */
115 #endif
116
117 #define US_PR_DEVICE    0xff          /* Use device's value */

```

说了这么多，U 盘属于其中的哪一种呢？USB 协议中规定，U 盘的 SubClass 是属于 `US_SC_SCSI` 的，而其通信协议使用的是 Bulk-Only 的，即这里看到的 `US_PR_BULK`。显然这些东西我们后来都会用得上。

那么这里还有一个 `match_flag`，它又是表示什么意思？`USB_INTERFACE_INFO` 这个宏好像把所有的设备的 `match_flag` 都给设成了 `USB_DEVICE_ID_MATCH_INT_INFO`，这是为什么？这个宏来自 `include/linux/usb.h`：

```

699 #define USB_DEVICE_ID_MATCH_INT_INFO \
700     (USB_DEVICE_ID_MATCH_INT_CLASS | \
701     USB_DEVICE_ID_MATCH_INT_SUBCLASS | \
702     USB_DEVICE_ID_MATCH_INT_PROTOCOL)

```

`match_flag` 这个东西是给 USB Core 去用的，USB Core 负责给设备寻找适合它的驱动，负责给驱动寻找适合它的设备，它所比较的就是 `struct usb_device_id` 的变量，而 `struct usb_device_id`



结构体中有许多成员，那么是不是一定要把每一个成员都给比较一下呢？其实就是告诉 USB Core，你只要比较 `bInterfaceClass`，`bInterfaceSubClass` 和 `bInterfaceProtocol` 即可。  
`include/linux/mod_devicetable.h` 中针对 `struct usb_device_id` 中的每一个要比较的项定义了一个宏：

```
122 /* Some useful macros to use to create struct usb_device_id */
123 #define USB_DEVICE_ID_MATCH_VENDOR          0x0001
124 #define USB_DEVICE_ID_MATCH_PRODUCT         0x0002
125 #define USB_DEVICE_ID_MATCH_DEV_LO         0x0004
126 #define USB_DEVICE_ID_MATCH_DEV_HI         0x0008
127 #define USB_DEVICE_ID_MATCH_DEV_CLASS      0x0010
128 #define USB_DEVICE_ID_MATCH_DEV_SUBCLASS   0x0020
129 #define USB_DEVICE_ID_MATCH_DEV_PROTOCOL   0x0040
130 #define USB_DEVICE_ID_MATCH_INT_CLASS      0x0080
131 #define USB_DEVICE_ID_MATCH_INT_SUBCLASS   0x0100
132 #define USB_DEVICE_ID_MATCH_INT_PROTOCOL   0x0200
```

回去对比一下 `struct usb_device_id` 就知道这些宏是什么意思了。

但是你一定有一个疑问，那就是为什么会有一个 `USUAL_DEV` 和一个 `UNUSUAL_DEV` 这样两个宏的存在，它们之间有什么区别呢？顾名思义，有些设备属于普通设备，而有些设备却并不是普通设备，它们或者是有一些别的设备不具备的特性，或者是他们遵循的通信协议有些与众不同，比如，它既不是 `Bulk-Only` 也不是 `CBI`，像这些不按常理出牌的设备，写代码的人把它们单独给列了出来。当然，从大的分类来看，它们依然是属于 `USB Mass Storage` 这个类别的，否则也没必要放在这个目录下面了。

为了包容这些另类的设备，伟大的 Linux 内核开发人员们为它们准备了一个文件，它就是让诸多 `USB Mass Storage` 设备厂家欢欣鼓舞的 `unusual_devs.h`，有了它，厂家们不用再为自己的设备不被 Linux 内核支持而烦恼了。

---

## 18. 冬天来了，春天还会远吗？（二）

打开 `unusual_devs.h` 吧，之前我们看了它的最后几行，但是如果你仔细看的话会发现最后几行和前面的一些行有着明显的不同。最后几行都是 `USUAL_DEV` 的宏，而前面则全是 `UNUSUAL_DEV` 的宏，随便看一下，发现每一行就是这么一个宏，毫无疑问它对应一种设备，我们从其中挑一个来看，比如挑一个三星的吧。

下面这个设备，正是来自三星的一个 `Flash` 产品。

```
1092 /* Submitted by Hartmut Wahl <hwahl@hwahl.de> */
1093 UNUSUAL_DEV( 0x0839, 0x000a, 0x0001, 0x0001,
1094             "Sa msung",
1095             "Digimax 410",
```

```

1096         US_SC_DEVICE, US_PR_DEVICE, NULL,
1097         US_FL_FIX_INQUIRY),

```

Digimax 410, 熟悉数码相机的人大概对三星的这款 410 万像素 3 倍光学变焦的产品不会陌生, 不过数码相机更新得很快, 这款在 2002 年推出的相机如今当然也算是很一般了, 市场上卖的话也就在 1500 元以下, 不过当时刚推出时也是 3000 元到 4000 元的。我们来看这一行是什么意思。

UNUSUAL\_DEV 这个宏的定义以及它所利用的 USB\_DEVICE\_VER 的定义我们前面都已经看到了。

所以这段对三星设备的描述的 UNUSUAL\_DEV 最终出现在 storage\_usb\_ids 中的意思就是令 match\_flags 为 USB\_DEVICE\_ID\_MATCH\_DEVICE\_AND\_VERSION, idVendor 为 0x0839, idProduct 为 0x000a, bcdDevice\_lo 为 0x0001, bcdDevice\_hi 为 0x0001。可是你会问了, 这里我们看到的这个 UNUSUAL\_DEV 里面有 10 个参数, 可是刚才我们看到的 USB\_DEVICE\_VER 却只能处理其中的前几个, 后面那些怎么办呢? 呵呵, 你可曾注意到, 在数组 storage\_usb\_ids 里面, 有如下两行:

```

141 #undef UNUSUAL_DEV
142 #undef USUAL_DEV

```

明明定义了这两个宏, 为什么又 undef 掉呢? 事实上, 在 storage\_usb\_ids 和 us\_unusual\_dev\_list 这两个数组之间, 我们又看到了下面这一段:

```

161 #define UNUSUAL_DEV(idVendor, idProduct, bcdDeviceMin, bcdDeviceMax, \
162         vendor_name, product_name, use_protocol, use_transport, \
163         init_function, Flags) \
164 { \
165     .vendorName = vendor_name,      \
166     .productName = product_name,    \
167     .useProtocol = use_protocol,     \
168     .useTransport = use_transport,   \
169     .initFunction = init_function,   \
170 }
171
172 #define USUAL_DEV(use_protocol, use_transport, use_type) \
173 { \
174     .useProtocol = use_protocol,     \
175     .useTransport = use_transport,   \
176 }

```

UNUSUAL\_DEV 和 USUAL\_DEV 竟然被定义了两次。这样对于三星产品来说, 这个宏的意思又是令 vendorName 为 “Sa msung”, 令 productName 为 “Digimax 410”, 而 useProtocol 为 US\_SC\_DEVICE, useTransport 为 US\_PR\_DEVICE, initFunction 为 NULL, flag 为 US\_FL\_FIX\_INQUIRY。

看明白了吗? 首先不去管各项的具体含义, 至少我们看出来, 针对同一个文件, 我们使用两次定义 UNUSUAL\_DEV 这个宏的方法, 两次利用了它的不同元素。换言之,

UNUSUAL\_DEV 这个宏本来可以设定 10 个参数，而 `storage_usb_ids` 中需要使用其中的前 4 个参数，同时 `us_unusual_dev_list` 中需要使用其中的后 6 个参数，所以在 `unusual_devs.h` 中定义的一行起了两个作用。这就是为什么不管是 `storage_usb_ids` 数组还是 `us_unusual_dev_list` 数组，其中都把 `unusual_devs.h` 文件给包含了进来。

而我们之所以使用两个数组的原因是，`storage_usb_ids` 是提供给 USB Core 的，它需要比较驱动和设备从而确定设备是被这个驱动所支持的，我们只需要比较四项就可以了，因为这四项已经足以确定一个设备的厂商、产品、序列号。比较这些就够了，而 `us_unusual_dev_list` 这个数组中的元素是我们接下来的代码要用的，比如它用什么协议，它有什么初始化函数，所以我们使用了两个数组。而我们需要注意的是，这两个数组中元素的顺序是一样的，所以我们从 `storage_usb_ids` 中得到的 `id_index` 用于 `us_unusual_dev_list` 也是可以的，表示的还是同一个设备。而这也就是我们刚才在 `get_device_info()->find_unusual()` 中看到的。

```
482 static struct us_unusual_dev *find_unusual(const struct usb_device_id *id)
483 {
484     const int id_index = id - storage_usb_ids;
485     return &us_unusual_dev_list[id_index];
486 }
```

看得出来，`id_index` 可以脚踏两只船，在 `storage_usb_ids` 和 `us_unusual_dev_list` 之间游刃有余。而 `find_unusual()` 这个函数的真实作用就是返回一个 `us_unusual_dev` 的指针。

```
494     struct us_unusual_dev *unusual_dev = find_unusual(id);
```

这个指针之后将被用到。

最后具体解释一下这行为三星这款数码相机写的语句。

(1) 关于 `match_flags`，它的值是 `USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION`。这是一个宏，它就告诉 USB Core，要比较这样几个方面，`idVendor`，`idProduct`，`bcdDevice_lo`，`bcdDevice_hi`，其中 `idVendor` 和下面的 `vendorName` 是对应的，而 `idProduct` 和下面的 `productName` 是对应的，业内为每家公司编一个号，这样便于管理，比如三星的编号就是 `0x0839`，那么三星的产品中就会在其设备描述符中 `idVendor` 的位上 `0x0839`。而三星自己的每种产品也会有一个编号，和 `Digimax 410` 对应的编号就是 `0x000a`，而 `bcdDevice_lo` 和 `bcdDevice_hi` 共同组成一个具体设备的编号(Device Release Number)，`bcd` 就意味着这个编号是二进制的格式。

(2) `.vendorName` 和 `productName` 为 “Sa msung” 和 “Digimax 410”。

(3) `.useProtocol` 为 `US_SC_DEVICE`，`useTransport` 为 `US_PR_DEVICE`，这种情况就说明对于这种设备，它属于什么 SubClass，它使用什么通信协议，得从设备描述符里面去读取，它都写在那里面了。一会儿会看到我们的代码中会如何判断这个的。

(4) `.initFunction` 等于 `NULL`，这个很有意义的，这个函数就是设备的初始化函数，一般

的设备都不需要这个函数，但是有些设备偏要标新立异，它就告诉你，要用我的设备必须先做一些初始化，于是它提供了一个函数，`initFunction`，当然这是一个函数指针，这里如果不为 `NULL` 的话，到时候就会被调用，以后我们会看到代码中对这个指针进行了判断。如果为空不理睬，否则就会执行。比如我们看下面两款设备，它们就分别提供了一个叫做 `usb_stor_sddr09_init()` 和 `usb_stor_sddr09_dpcm_init()` 的初始化函数，

```

422 #ifdef CONFIG_USB_STORAGE_SDDR09
423 UNUSUAL_DEV( 0x04e6, 0x0003, 0x0000, 0x9999,
424             "Sandisk",
425             "ImageMate SDDR09",
426             US_SC_SCSI, US_PR_EUSB_SDDR09, usb_stor_sddr09_init,
427             0),
428
429 /* This entry is from Andries.Brouwer@cw.nl */
430 UNUSUAL_DEV( 0x04e6, 0x0005, 0x0100, 0x0208,
431             "SCM Microsystem",
432             "eUSB SmartMedia / CompactFlash Adapter",
433             US_SC_SCSI, US_PR_DPCM_USB, usb_stor_sddr09_dpcm_init,
434             0),
435 #endif

```

(5) `flag` 等于 `US_FL_FIX_INQUIRY`，这个 `flag` 可以设为很多值，它的存在本身就表示这个设备有一些与众不同，因为一般的设备是不用这个 `flag` 的，有这个 `flag` 就表明这个设备可能在某些地方需要进行特殊的处理，所以今后在代码中我们会看到突然跳出一句，判断 `us->flag` 等于某个数值，如果等于，就执行一些代码；如果不等于，那就不做任何事情。这个 `flag` 的存在也使得我们可以方便处理一些设备的 `Bug`，比如正常的设备你问它：“吃了吗？”它就回答：“吃了”。可是不正常的设备可能会根本不回答，或者回答：“北京房价真高！”于是对于这种设备，可能我们就需要一些专门的代码来对付。具体到这个 `US_FL_FIX_INQUIRY`，这个 `flag` 这么一设置，就表明这个设备在接收到 `INQUIRY` 命令时会有一些异常的特征，所以以后我们会在代码里看到我们是如何处理它的。设置了这个 `flag` 的当然不止是三星的这款相机，别的设备也有可能设置的。

(6) 既然明白了 `unusual_devs.h` 的作用，那么要做的是很显然的一件事情，如果一个厂家推出了一个新的设备，它有一些新的特征，而目前的设备驱动不足以完全支持它，那么厂家首先需要做的事情就是在 `unusual_devs.h` 中添加一个 `UNUSUAL_DEV` 来定义自己的设备，然后再看是否需要给内核打补丁以及如何打。因此这几年 `unusual_devs.h` 这个文件的长度也是慢慢在增长。

## 19. 冬天来了，春天还会远吗？（三）

从两张表得到了我们需要的东西，然后下面的代码就是围绕着这两个指针来展开了。

(unusual\_dev 和 id)继续看 get\_device\_info()。

497 行，把 unusual\_dev 给记录在 us 里面，反正 us 里面也有这么一个成员。这样记录下来以后使用起来就方便了，因为 us 是贯穿整个故事的，所以访问它的成员很方便，随时都可以，但是 us\_unusual\_dev\_list 以及 storage\_usb\_ids 这两张表这次之后就不会再用了。因为我们已经得到了我们想要的，所以就不用再去骚扰这两个数组了。

498 行至 504 行，给 us 的另外三个成员赋值，subclass、protocol 和 flags。比如我们的 U 盘，它属于主流设备，在 us\_unusual\_dev\_list 列表中能找到它，其 subclass 是 US\_SC\_SCSI，而 protocol 是 Bulk-only，即这里用宏 US\_PR\_BULK 代表。

关于 US\_SC\_DEVICE 和 US\_PR\_DEVICE 在之前讲三星的数码相机时已经看到了，它就表示 subclass 和 protocol 得从设备的描述符里面读出来。这样做看起来很滑稽，因为三星完全可以把 subclass 和 protocol 在 UNUSUAL\_DEV 中写清楚，何必让我们再去读设备描述符呢？然而，我们可以想象，这样的好处是定义一个 UNUSUAL\_DEV 可以代表几种设备，即它可以让几个不同 subclass 的设备共用这么一个宏，或者几个不同 protocol 的设备共用这么一个宏。需要特别指出的是 us->flags，对于 U 盘来说，它当然没有什么 flags 需要设定，但是 unusual\_devs.h 中的很多设备都设置了各种 flags，稍后在代码中我们会看到，时不时就得判断一下是否某个 flag 设置了，通常是如果设置了，就要多执行某段代码，以满足某种要求。

523 行到 551 行，这是一段纯粹的调试代码，对我们理解 USB 没有任何意义的。这段代码检查 unusual\_devs.h，看是否这个文件定义了一行没有意义的句子。什么叫没有意义？我们刚才看见了，如果这个设备设了 US\_SC\_DEVICE，那么其 subclass 将从描述符中读出来，如果不然，则让 subclass=unusual\_dev->useProtocol，但是如果后者又真的和描述符里读出来的一样，那么这个设备就没有必要把自己的 useProtocol 定义在 unusual\_devs.h 中了，因为反正也可以从描述符里读出来。还不如和大众一样设为 US\_SC\_DEVICE 得了。就比如我们来看下面这行代表一个 Sony 的 Memory Stick 产品的代码：

```
629 UNUSUAL_DEV( 0x054c, 0x0069, 0x0000, 0x9999,  
630             "Sony",  
631             "Memorystick MSC-U03",  
632             US_SC_UFI, US_PR_CB, NULL,  
633             US_FL_SINGLE_LUN ),
```

我们看到其 useProtocol 这一栏里写了 US\_SC\_UFI，这表明它自称是属于 UFI 这个 SubClass 的，但是如果我们从它的描述符里面读出来也是这个，那就没有必要注明在这里了，这里直接写成 US\_SC\_DEVICE 好了。当然，总的来说这段代码有一些傻。写代码的是希望能够更好地管理 unusual\_devs.h，希望它不要不断增加，它总希望能够从这个文件中删除一些行，并且即使不能删除一行，也希望每一行都看上去整齐一点，让这个文件看上去更加小巧玲珑，更加精致。而不是无休地增加，不息地扩充。

至此，get\_device\_info 这个函数就结束了它的使命。在 USB Storage 这部戏里，它将不再

出场。但我想说，对于 USB Storage 这个整个模块来说，主角配角不重要，每个函数都是画布上的一抹色彩。就像我们每一个人，不也是别人人生中的配角，但总是自己人生的主角吗？

## 20. 冬天来了，春天还会远吗？（四）

结束了 `get_device_info`，我们继续沿着 `storage_probe` 一步一步地走下去。继续，这就是我们前面提到过的三个函数，`get_transport`、`get_protocol` 和 `get_pipes`。一旦结束了这三个函数，我们就将进入本故事的高潮部分。而在这之前，我们只能一个一个地来看。好在这几个函数虽然不短，但是真正有用的信息只有一点，所以可以很快看完。

```

993     /* Get the transport, protocol, and pipe settings */
994     result = get_transport(us);
995     if (result)
996         goto BadDevice;
997     result = get_protocol(us);
998     if (result)
999         goto BadDevice;
1000    result = get_pipes(us);
1001    if (result)
1002        goto BadDevice;

```

第 1 个，`get_transport(us)`。

```

557 static int get_transport(struct us_data *us)
558 {
559     switch (us->protocol) {
560     case US_PR_CB:
561         us->transport_name = "Control/Bulk";
562         us->transport = usb_stor_CB_transport;
563         us->transport_reset = usb_stor_CB_reset;
564         us->max_lun = 7;
565         break;
566
567     case US_PR_CBI:
568         us->transport_name = "Control/Bulk/Interrupt";
569         us->transport = usb_stor_CBI_transport;
570         us->transport_reset = usb_stor_CB_reset;
571         us->max_lun = 7;
572         break;
573
574     case US_PR_BULK:
575         us->transport_name = "Bulk";
576         us->transport = usb_stor_Bulk_transport;
577         us->transport_reset = usb_stor_Bulk_reset;
578         break;
579
580 #ifdef CONFIG_USB_STORAGE_USBAT
581     case US_PR_USBAT:
582         us->transport_name = "Shuttle USBAT";
583         us->transport = usb_at_transport;

```

```

584         us->transport_reset = usb_stor_CB_reset;
585         us->max_lun = 1;
586         break;
587 #endif
588
589 #ifdef CONFIG_USB_STORAGE_SDDR09
590     case US_PR_EUSB_SDDR09:
591         us->transport_name = "EUSB/SDDR09";
592         us->transport = sddr09_transport;
593         us->transport_reset = usb_stor_CB_reset;
594         us->max_lun = 0;
595         break;
596 #endif
597
598 #ifdef CONFIG_USB_STORAGE_SDDR55
599     case US_PR_SDDR55:
600         us->transport_name = "SDDR55";
601         us->transport = sddr55_transport;
602         us->transport_reset = sddr55_reset;
603         us->max_lun = 0;
604         break;
605 #endif
606
607 #ifdef CONFIG_USB_STORAGE_DPCM
608     case US_PR_DPCM_USB:
609         us->transport_name = "Control/Bulk-EUSB/SDDR09";
610         us->transport = dpcm_transport;
611         us->transport_reset = usb_stor_CB_reset;
612         us->max_lun = 1;
613         break;
614 #endif
615
616 #ifdef CONFIG_USB_STORAGE_FREECOM
617     case US_PR_FREECOM:
618         us->transport_name = "Freecom";
619         us->transport = freecom_transport;
620         us->transport_reset = usb_stor_freecom_reset;
621         us->max_lun = 0;
622         break;
623 #endif
624
625 #ifdef CONFIG_USB_STORAGE_DATAFAB
626     case US_PR_DATAFAB:
627         us->transport_name = "Datafab Bulk-Only";
628         us->transport = datafab_transport;
629         us->transport_reset = usb_stor_Bulk_reset;
630         us->max_lun = 1;
631         break;
632 #endif
633
634 #ifdef CONFIG_USB_STORAGE_JUMPSHOT
635     case US_PR_JUMPSHOT:
636         us->transport_name = "Lexar Jumpshot Control/Bulk";
637         us->transport = jumpshot_transport;
638         us->transport_reset = usb_stor_Bulk_reset;
639         us->max_lun = 1;
640         break;
641 #endif
642

```

```

643 #ifdef CONFIG_USB_STORAGE_ALAUDA
644     case US_PR_ALAUDA:
645         us->transport_name = "Alauda Control/Bulk";
646         us->transport = alauda_transport;
647         us->transport_reset = usb_stor_Bulk_reset;
648         us->max_lun = 1;
649         break;
650 #endif
651
652 #ifdef CONFIG_USB_STORAGE_KARMA
653     case US_PR_KARMA:
654         us->transport_name = "Rio Karma/Bulk";
655         us->transport = rio_karma_transport;
656         us->transport_reset = usb_stor_Bulk_reset;
657         break;
658 #endif
659
660     default:
661         return -EIO;
662 }
663 US_DEBUGP("Transport: %s\n", us->transport_name);
664
665 /* fix for single-lun devices */
666 if (us->flags & US_FL_SINGLE_LUN)
667     us->max_lun = 0;
668 return 0;
669 }

```

乍一看，这么长一段，不过明眼人一看就知道了，主要就是一个 **switch**，选择语句，语法上来说很简单，所以我们看懂这段代码不难。只是，我想说的是，虽然这里做出一个选择不难，但是不同选择就意味着后来整个故事会有千差万别的结局，当鸟儿选择在两翼上系上黄金，就意味着它放弃展翅高飞；选择云天搏击，就意味着放弃身外的负累。

所以，此处，我们需要仔细地看清楚我们究竟选择了怎样一条路。很显然，前面我们已经说过，对于 U 盘，spec 规定了，它就属于 **Bulk-only** 的传输方式，即它的 `us->protocol` 就是 **US\_PR\_BULK**。这是我们刚刚在 `get_device_info` 中确定下来的。于是，在整个 **switch** 段落中，我们所执行的只是 **US\_PR\_BULK** 这一段，即 `us` 的 `transport_name` 被赋值为“Bulk”，`transport` 被赋值为 `usb_stor_Bulk_transport`，`transport_reset` 被赋值为 `usb_stor_Bulk_reset`。其中我们最需要记住的是，`us` 的成员 `transport` 和 `transport_reset` 是两个函数指针。程序员们把这个称作“钩子”。这两个赋值我们需要牢记，日后我们一定会用到它们，因为这正是我们真正的数据传输时调用的东西。关于 `usb_stor_Bulk_*` 的这两个函数，到时候调用了再来看。现在只需知道，日后我们一定会回过来看这个赋值的。

580 行到 658 行，不用多说了，这里全是与各种特定产品相关的一些编译开关，它们有一些自己定义一些传输函数，有些则共用通用的函数。

666 行，判断 `us->flags`，还记得我们在讲 `unusual_devs.h` 文件时说的 `flags` 吧，这里第一次用上了。有些设备设置了 **US\_FL\_SINGLE\_LUN** 这个 `flag`，就表明它是只有一个 LUN 的。像这



样的设备挺多的，随便从 `unusual_devs.h` 中抓一个出来：

```
596 UNUSUAL_DEV( 0x054c, 0x002d, 0x0100, 0x0100,  
597             "Sony",  
598             "Memorystick MSAC-US1",  
599             US_SC_DEVICE, US_PR_DEVICE, NULL,  
600             US_FL_SINGLE_LUN ),
```

比如这个 Sony 的 Memorystick。中文名叫“记忆棒”，大小就与口香糖一样，也是一种存储芯片。它是 Sony 公司推出的，广泛用于 Sony 的各种数码产品中，比如数码照相机、数码摄影机。

LUN 就是 Logical Unit Number。通常在谈到 SCSI 设备时不可避免要说起 LUN。关于 LUN，曾几何时，一位来自 Novell 的参与开发 Linux 内核中 USB 子系统的工程师这样对我说，一个 LUN 就是设备中的一个驱动。下面举例来说一下 USB 中引入 LUN 的目的。有些读卡器可以有多个插槽，比如有两个，其中一个支持 CF 卡，另一个支持 SD 卡，那么这种情况要区分这两个插槽里的设备，就得引入 LUN 有这个概念，即逻辑单元。很显然，像 U 盘这样简单的设备其 LUN 必然是一个。有时候，人们常把 U 盘中一个分区当做一个 LUN，但是不应该这么理解。

知道了 LUN 以后，自然就可以知道 `US_FL_SINGLE_LUN` 是做什么了，这个 flag 的意义很明显，直截了当地告诉你，这个设备只有一个 LUN，它不支持多个 LUN。而 `max_lun` 又是什么意思？`us` 中的成员 `max_lun` 等于一个设备所支持的最大 LUN 号。即如果一个设备支持四个 LUNs，那么这四个 LUN 的编号就是 0, 1, 2, 3，而 `max_lun` 就是 3。如果一个设备不用支持多个 LUN，那么它的 `max_lun` 就是 0。所以这里 `max_lun` 就是设为 0。

另外一个需要注意的地方是，比较一下各个 case 语句会发现 `US_PR_BULK` 和其他的 case 不一样，其他的 case 下面都设置了 `us->max_lun`，而对应于 Bulk-Only 协议的这个 case，它没有设置 `us->max_lun`，之所以不设，是因为这个值由设备说了算，必须向设备查询，这是 Bulk-Only 协议规定的。在 `drivers/usb/storage/transport.c` 中定义了一个 `usb_stor_Bulk_max_lun()` 函数，它将负责获取这个 `max lun`。而我依然要声明一次，这个函数对我们 U 盘没有什么意义，这个值肯定是 0，所以这个函数咱们就不去理睬了。

至此，`get_transport()` 也结束了，和 `get_device_info()` 一样。我们目前所看到的这些函数都不得不面对现实，对它们来说，凋谢是最终的结果，盛开只是一个过程。而对我们来说，要到达终点，那么和这些函数狭路相逢，终不能幸免。然而，不管这部分代码有多么重要，也不过是我们整个长途旅程中，来去匆匆的转车站，无论停留多久，始终要离去坐另一班机。

## 21. 冬天来了，春天还会远吗？（五）

看完了 `get_transport()` 继续看 `get_protocol()` 函数和 `get_pipes()` 函数。仍然是来自 `drivers/usb/storage/usb.c` 中：

```

672 static int get_protocol(struct us_data *us)
673 {
674     switch (us->subclass) {
675     case US_SC_RBC:
676         us->protocol_name = "Reduced Block Commands (RBC)";
677         us->proto_handler = usb_stor_transparent_scsi_command;
678         break;
679
680     case US_SC_8020:
681         us->protocol_name = "8020i";
682         us->proto_handler = usb_stor_ATAPI_command;
683         us->max_lun = 0;
684         break;
685
686     case US_SC_QIC:
687         us->protocol_name = "QIC-157";
688         us->proto_handler = usb_stor_qic157_command;
689         us->max_lun = 0;
690         break;
691
692     case US_SC_8070:
693         us->protocol_name = "8070i";
694         us->proto_handler = usb_stor_ATAPI_command;
695         us->max_lun = 0;
696         break;
697
698     case US_SC_SCSI:
699         us->protocol_name = "Transparent SCSI";
700         us->proto_handler = usb_stor_transparent_scsi_command;
701         break;
702
703     case US_SC_UFI:
704         us->protocol_name = "Uniform Floppy Interface (UFI)";
705         us->proto_handler = usb_stor_ufi_command;
706         break;
707
708 #ifdef CONFIG_USB_STORAGE_ISD200
709     case US_SC_ISD200:
710         us->protocol_name = "ISD200 ATA/ATAPI";
711         us->proto_handler = isd200_ata_command;
712         break;
713 #endif
714
715     default:
716         return -EIO;
717     }
718     US_DEBUGP("Protocol: %s\n", us->protocol_name);
719     return 0;
720 }

```

这段代码非常浅显易懂。根据 `us->subclass` 来判断。对于 U 盘来说，`spec` 里面规定了，它

的 SubClass 是 US\_SC\_SCSI，所以这里就是两句赋值语句：一个是令 us 的 protocol\_name 为 “Transparent SCSI”，另一个是令 us 的 proto\_handler 为 usb\_stor\_transparent\_scsi\_command，后者又是一个函数指针。

然后是 get\_pipes()，来自 drivers/usb/storage/usb.c:

```

723 static int get_pipes(struct us_data *us)
724 {
725     struct usb_host_interface *altsetting =
726         us->pusb_intf->cur_altsetting;
727     int i;
728     struct usb_endpoint_descriptor *ep;
729     struct usb_endpoint_descriptor *ep_in = NULL;
730     struct usb_endpoint_descriptor *ep_out = NULL;
731     struct usb_endpoint_descriptor *ep_int = NULL;
732
733     /*
734      * Find the first endpoint of each type we need.
735      * We are expecting a minimum of 2 endpoints - in and out (bulk).
736      * An optional interrupt-in is OK (necessary for CBI protocol).
737      * We will ignore any others.
738      */
739     for (i = 0; i < altsetting->desc.bNumEndpoints; i++) {
740         ep = &altsetting->endpoint[i].desc;
741
742         if (usb_endpoint_xfer_bulk(ep)) {
743             if (usb_endpoint_dir_in(ep)) {
744                 if (!ep_in)
745                     ep_in = ep;
746             } else {
747                 if (!ep_out)
748                     ep_out = ep;
749             }
750         }
751
752         else if (usb_endpoint_is_int_in(ep)) {
753             if (!ep_int)
754                 ep_int = ep;
755         }
756     }
757
758     if (!ep_in || !ep_out || (us->protocol == US_PR_CBI && !ep_int)) {
759         US_DEBUGP("Endpoint sanity check failed! Rejecting dev.\n");
760         return -EIO;
761     }
762
763     /* Calculate and store the pipe values */
764     us->send_ctrl_pipe = usb_sndctrlpipe(us->pusb_dev, 0);
765     us->recv_ctrl_pipe = usb_rcvctrlpipe(us->pusb_dev, 0);
766     us->send_bulk_pipe = usb_sndbulkpipe(us->pusb_dev,
767         ep_out->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
768     us->recv_bulk_pipe = usb_rcvbulkpipe(us->pusb_dev,
769         ep_in->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
770     if (ep_int) {
771         us->recv_intr_pipe = usb_rcvintpipe(us->pusb_dev,
772             ep_int->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
773         us->ep_bInterval = ep_int->bInterval;

```

```

774     }
775     return 0;
776 }

```

这个函数应该可以说是比较复杂的一个。请容我慢慢给您道来。

726 行, `us->pusb_intf`, 可还记得, 在 `associate_dev` 中赋的值, 如不记得请回过去查一下。没错, `us->pusb_intf` 就是我们故事中最开始一再提到的 `interface` (指针)。而它的成员 `cur_altsetting`, 就是当前的设置。在讲 `associate_dev` 时也已经遇到过, 它是一个 `struct usb_host_interface` 的结构体指针。现在这里用另一个指针临时代替一下, `altsetting`。接下来会用到它的成员: `desc` 和 `endpoint`。

回顾 `struct usb_host_interface`, 可以看到它两个成员: `struct usb_interface_descriptor desc` 和 `struct usb_host_endpoint *endpoint`。其中, `desc` 不用多说, 正是这个接口的接口描述符, 而 `endpoint` 这个指针记录的是几个端点, 它们以数组的形式被存储, 而 `endpoint` 指向数组头。这些都是在 USB Core 枚举时就设置好了, 我们无需操任何心, 只需拿来使用就是了。这里给出 `struct usb_host_endpoint` 的定义, 来自 `include/linux/usb.h`:

```

59 struct usb_host_endpoint {
60     struct usb_endpoint_descriptor desc;
61     struct list_head      urb_list;
62     void                  *hcpriv;
63     struct ep_device      *ep_dev;          /* For sysfs info */
64
65     unsigned char *extra; /* Extra descriptors */
66     int extralen;
67 };

```

接着定义了几个 `struct usb_endpoint_descriptor` 的结构体指针。顾名思义, 这就是对应端点的描述符。其定义来自于 `include/linux/usb/ch9.h`:

```

312 /* USB_DT_ENDPOINT: Endpoint descriptor */
313 struct usb_endpoint_descriptor {
314     __u8 bLength;
315     __u8 bDescriptorType;
316
317     __u8 bEndpointAddress;
318     __u8 bmAttributes;
319     __le16 wMaxPacketSize;
320     __u8 bInterval;
321
322     /* NOTE: these two are _only_ in audio endpoints. */
323     /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324     __u8 bRefresh;
325     __u8 bSynchAddress;
326 } __attribute__((packed));

```

至此, 四大描述符一一亮相, 在继续讲之前, 我们先来小结一下: 究竟什么是描述符? 每个 USB 设备都有四大描述符, 这里拿 U 盘来举例。听说过 Flash Memory 吗? Intel、三星, 这些都是做 Flash Memory 的, 当然通常人们就简称 Flash。Flash 在 U 盘中扮演什么角色? Flash

是用来给用户存储数据的，而 U 盘中的 Flash 就相当于 PC 中的硬盘，存储数据主要就靠它。那么除了给用户存储数据以外，设备自己还需要存储一些设备本身固有的信息，比如设备姓名谁？谁生产的？还有一些信息，比如该设备有几种配置，有几个接口，等许多特性。

这个世界上，除了 Flash memory 外，还有一个东西叫做 EEPROM，也是用来存储的，它是 EEPROM 的前身，而 Flash 是基于 EEPROM 技术发展起来的一种低成本的 ROM 产品。

EEPROM 和 Flash 相同，都是需要电擦除，但 EEPROM 可以按字节擦除，而不像 Flash 那样一次擦除一个 block，这样在只需改动很少数据的情况下使用 EEPROM 就很方便了。因此 EEPROM 的这一特性使它的电路要复杂一些，并且集成度不高，一个 bit 需要两个管子：一个用来储存电荷信息，一个充当开关。所以 EEPROM 的成本高，Flash 简化了一些电路，成本降低了很多。

因此，在 USB 设备中通常会有一个 Flash 芯片，以及一个 EEPROM 芯片。Flash 用于为客户存储数据，而 EEPROM 用来存储设备本身的信息。这就是为什么当 Intel 把 Flash 芯片卖给摩托罗拉之后，客户看到的手机厂商是摩托罗拉而不是 Intel，因为我们虽然在做 Flash 时把我们的厂商 ID 写在了 Flash 上，但是对于最终的成品对外来看，提供的信息都是来自 EEPROM，所以当你把 USB 设备通过 USB 接口连到电脑上去，电脑上如果能显示厂家，那么一定是最终的包装厂家，而不可能是那块 Flash 的厂家。而 EEPROM 里边写什么？按什么格式写？这正是 USB spec 规定的，这种格式就是一个个描述符的格式。设备描述符、配置描述符、接口描述符、端点描述符，以及其他一些某一些类别的设备特有的描述符，比如 Hub 描述符都是很规范的，尤其对于这四种标准的描述符，每个 USB 设备都是规规矩矩地支持的。所以 USB Core 层可以用一段相同的代码把它们都给读出来，而不用再让我们设备驱动程序去自己读了，这就是权力集中的好处，反正大家都要做的事情，干脆让上头一起做好了。

739 行到 756 行，循环，bNumEndpoints 就是接口描述符中的成员，表示这个接口有多少个端点，不过这其中包括 0 号端点，0 号端点是任何一个 USB 设备都必须是提供的，这个端点专门用于进行控制传输，即它是一个控制端点。正因为如此，所以即使一个设备没有进行任何设置，USB 主机也可以开始跟它进行一些通信，因为即使不知道其他端点，但至少知道它一定有一个 0 号端点，或者说一个控制端点。

此外，通常 USB Mass Storage 会有两个批量端点，用于批量传输，即所谓的批量传输。我们日常的读写 U 盘里的文件，就是属于批量传输。所以毫无疑问，对于 Mass Storage 设备来说，批量传输是它的主要工作方式，道理很简单，我们使用 U 盘就是用来读写文件的。和这些描述符打交道无非就是为了帮助我们最终实现读写文件的工作，这才是每一个 USB 存储设备真正的使命。

于是我们来看这段循环到底在干什么，altsetting->endpoint[i].desc，对照 struct usb\_host\_endpoint 这个结构体的定义可知，desc 正是一个 struct usb\_endpoint\_descriptor 的变量。

刚刚定义了四个这种结构体的指针，ep，ep\_in，ep\_out 和 ep\_int，很简单，就是用来记录端点描述符的，ep\_in 用于 Bulk-IN，ep\_out 用于 Bulk-OUT，ep\_int 用于记录中断端点（如果有的话）。而 ep，只是一个临时指针。

我们看 struct usb\_endpoint\_descriptor，在它的成员中，bmAttributes 表示属性，总共 8 位，其中 bit1 和 bit0 共同称为 Transfer Type，即传输类型，00 表示控制，01 表示等时，10 表示批量，11 表示中断。因此通过比较这些成员，就可以判断该端点的传输类型，这个函数 usb\_endpoint\_xfer\_bulk() 定义于 include/linux/usb.h 中：

```
571 static inline int usb_endpoint_xfer_bulk(const struct usb_endpoint_
descriptor *epd)
572 {
573     return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
574             USB_ENDPOINT_XFER_BULK);
575 }
```

而 USB\_ENDPOINT\_XFERTYPE\_MASK 这个宏定义于 include/linux/usb/ch9.h 中：

```
338 #define USB_ENDPOINT_XFERTYPE_MASK    0x03    /* in bmAttributes */
339 #define USB_ENDPOINT_XFER_CONTROL      0
340 #define USB_ENDPOINT_XFER_ISOC         1
341 #define USB_ENDPOINT_XFER_BULK         2
342 #define USB_ENDPOINT_XFER_INT          3
```

从上面的注释可以看出，742 行就是判断这个端点描述符描述的是不是一个批量端点。如果是，继续比较。我们先看 bEndpointAddress，这个 struct usb\_endpoint\_descriptor 中的另一个成员，也是 8 个 bit，或者说 1 个 Byte，其中 bit7 表示这个端点的方向，0 表示 OUT，1 表示 IN。OUT 与 IN 是对主机而言，OUT 就是从主机到设备，IN 就是从设备到主机。因此比较这个成员就可以判断端点的方向，干这件事的函数 usb\_endpoint\_dir\_in() 也来自 include/linux/usb.h：

```
549 static inline int usb_endpoint_dir_in(const struct usb_endpoint_
descriptor *epd)
550 {
551     return ((epd->bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_IN);
552 }
```

而宏 USB\_DIR\_IN 和 USB\_ENDPOINT\_DIR\_MASK 仍然来自 include/linux/usb\_ch9.h：

```
48 #define USB_DIR_OUT                    0        /* to device */
49 #define USB_DIR_IN                      0x80    /* to host */
336 #define USB_ENDPOINT_DIR_MASK          0x80
```

所以这里意思很明显，就是为了让 ep\_in 和 ep\_out 指向该指的 endpoint descriptor。

而接下来 752 行，usb\_endpoint\_is\_int\_in() 来自 include/linux/usb.h：

```
646 static inline int usb_endpoint_is_int_in(const struct usb_endpoint_
descriptor *epd)
647 {
648     return (usb_endpoint_xfer_int(epd) && usb_endpoint_dir_in(epd));
649 }
```

有了前面两个函数作为基础，这个函数就不用再说了。总之，752 行中 `else if` 的作用就是如果这个端点是中断端点，那么就让 `ep_int` 指向它。我们说了，每一类 USB 设备其上面有多少端点有何种端点都是不确定的，都得遵守该类设备的规范，而 USB Mass Storage 的规范中规定了，一个 USB Mass Storage 设备至少应该有两个批量端点，控制端点显然是必需的。毋庸置疑，另外，可能会有一个中断端点，这种设备支持 CBI 协议，即 Control/Bulk/Interrupt 协议。我们也说过了，U 盘遵守的是 Bulk-only 协议，它不需要有中断端点。

758 行到 761 行这段代码，没什么好说的，就是判断 `ep_in` 或者 `ep_out` 是否存在，或者是遵守 CBI 协议但是没有中断端点，这些都是不合理的，当然就会出错！

剩下一小段代码，我们下节再看。需要说的是，这个函数结束之后我们将开始最精彩的部分，它就是伟大的 `usb_stor_acquire_resources()`。黑暗即将过去，黎明已经带我们上路。让我们共同期待吧。同时，我们小结一下，此前我们花了很大的篇幅来为 `usb_stor_acquire_resources()` 做铺垫，那我们来回顾一下，它究竟做了哪些事情？

首先我们从 `storage_probe` 出发，一共调用了五个函数，它们是 `assocait_dev`，`get_device_info`，`get_transport`，`get_protocol`，`get_pipes`。我们这样做的目的是什么？很简单，就是为了建立一个数据结构，它就是传说中的 `struct us_data`，它的名字叫做 `us`。我们把它建立了起来，为它申请了内存，为它的各个元素赋了值，目的就是为了让以后我们可以很好地利用它。

这五个函数都不难，你一定也会写。难的是如何去定义 `struct us_data`，别忘了这个数据结构是写代码的同志们专门为 `usb-storage` 模块而设计的。所谓编程，无非就是数据结构加上算法。没错，这个定义于 `drivers/usb/storage/usb.h` 中的数据结构长达 60 行，关于她的成员，我们还有很多没遇到，不过别急，后面会遇到的。好了，虽然 `get_pipes` 还有一小段没讲，但是我们可以提前和这 5 个函数说再见了，席慕蓉说过，若不得不分离，也要好好地说声再见，也要在心里存着一份感谢，谢谢她给你一份记忆。

---

## 22. 通往春天的管道

1991 年，一个在 Linux 中引入了管道这个概念，并且把管道用在很多地方，如文件系统、设备驱动中。于是后来我们看到在 Linux 中有了各种各样的管道。但是相同的是，所有管道都是用来传输东西的，只不过有些管道传输的是实实在在的物质，而有些管道传输的是数据。

眼下我们在 USB 代码中看到的管道就是用来传输数据及通信。通信是双方的，不可能自言自语。而在 USB 的通信中，一方肯定是主机，另一方是什么？是设备吗？说得更确切一点，

真正和主机进行通信的是设备内的端点。关于端点，我们也可以专业一点说，从硬件上来看它是实实在在存在的，它被实现为一种 FIFO，支持多少个端点是接口芯片的一个重要指标。

而从概念上来说，端点是主机和 USB 设备之间通信流的终点。主机和设备可以进行不同种类的通信，或者说数据传输。首先，设备连接在 USB 总线上，USB 总线为了分辨每一个设备，给每一个设备编上号，然后为了实现多种通信，设备方于是提供了端点，端点多了，自然也要编上号，而让主机最终和端点去联系。

在 USB 的世界里，有四种管道，它们对应 USB 世界中的四种传输方式：控制传输对应控制管道、中断传输对应中断管道、批量传输对应批量管道及等时传输对应等时管道。每一个 USB 世界中的设备要在 USB 世界里生存，就得有它生存的管道。而管道的出现，正是为了让我们分辨端点，或者说连接端点。记得网友“唐伯虎点蚊香”曾经说过：“主机与端点之间的数据链接就称为管道。”

比如说，复旦大学有一个主校区，也可以说是教学区，（当然也包括一些试验室），上课什么的都得去教学区，把教学区看做主机，那么与其相对的是，另外有很多学生宿舍楼，宿舍楼多了，就给每个楼编上号，比如 1 号楼，2 号楼，……，36 号楼，……，每幢楼算一个设备。

复旦主校区是主机，每幢宿舍楼算一个设备，你住的那间宿舍就算端点。那么管道呢？管道很难与现实中的某个实物对应，不能说它是复旦正门通往宿舍的某条路，而应该按别的方式理解。它包含很多东西，你可以把它比做快递的货物上面贴得那张标签，比如它上面写了收货人的地址，包括多少号楼多少号房，在 USB 里面，就是设备号和端点号，知道了这两个号，货物就能确定它的目的地，而 USB 主机就能知道和它通信的是哪个端点。

而管道除了包含着两个号以外，还包含其他一些信息。比如它包含了通信的方向，也就是说，你能从那张标签上得知 36 号楼 201 这个是收件人呢还是寄件人，虽然现实中不需要写明，因为快递员肯定知道你是收件人。管道里还包含了管道的类型，比如你的快递是从深圳递过来，那么怎么传递就得看快递公司了，快递公司肯定提供不同类型的服务，有的快有的慢，有的可能还有保险，看你出多少钱，让你选择不同的服务类型。同样管道也如此，因为 USB 设备的端点有不同的类型，所以管道里就包含了一个字段来记录这一点。好，让我们来查看实际的管道吧。

来看这些宏，头一个闪亮登场的是 `usb_sndctrlpipe`，定义于 `include/linux/usb.h` 中，先把这一堆东西相关的代码给列出来：

```
1432 static inline unsigned int __create_pipe(struct usb_device *dev,
1433         unsigned int endpoint)
1434 {
1435     return (dev->devnum << 8) | (endpoint << 15);
1436 }
1437
1438 /* Create various pipes... */
1439 #define usb_sndctrlpipe(dev, endpoint) \
```



```

1440      ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint))
1441 #define usb_rcvctrlpipe(dev,endpoint) \
1442      ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1443 #define usb_sndisocpipe(dev,endpoint) \
1444      ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint))
1445 #define usb_rcvisocpipe(dev,endpoint) \
1446      ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1447 #define usb_sndbulkpipe(dev,endpoint) \
1448      ((PIPE_BULK << 30) | __create_pipe(dev,endpoint))
1449 #define usb_rcvbulkpipe(dev,endpoint) \
1450      ((PIPE_BULK << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1451 #define usb_sndintpipe(dev,endpoint) \
1452      ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint))
1453 #define usb_rcvintpipe(dev,endpoint) \
1454      ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1455

```

先看 1439 行，把这个宏展开，PIPE\_CONTROL 也是宏，定义于同一个文件中，include/linux/usb.h:

```

1405 /* NOTE: these are not the standard USB_ENDPOINT_XFER_* values!! */
1406 /* (yet ... they're the values used by usbfs) */
1407 #define PIPE_ISOCHRONOUS      0
1408 #define PIPE_INTERRUPT       1
1409 #define PIPE_CONTROL         2
1410 #define PIPE_BULK            3

```

USB 有四种传输方式，即等时传输、中断传输、控制传输和批量传输。一个设备能支持这四种传输中的哪一种或者哪几种是设备本身的属性，在硬件设计时就确定了。比如一个纯粹的 U 盘，它肯定是支持批量传输和控制传输的。不同的传输要求有不同的端点，所以对于 U 盘来说，它一定会有批量端点和控制端点，于是就得使用相应的管道来跟不同的端点联系。在这里我们看到了四个宏，其中，PIPE\_ISOCHRONOUS 就是标志等时通道，PIPE\_INTERRUPT 就是中断通道，PIPE\_CONTROL 就是控制通道，PIPE\_BULK 就是批量通道。

另外\_\_create\_pipe 也是一个宏，由上面的定义可以看出它为构造一个宏提供了设备号和端点号。在内核中使用一个 unsigned int 类型的变量来表征一个管道，其中 8 位~14 位是设备号，即 devnum，15 位~18 位是端点号，即 endpoint。而咱们还看到有这么一个宏，USB\_DIR\_IN，是用来在管道里面标志数据传输方向的，一个管道要么只能输入，要么只能输出，鱼和熊掌不可兼得也，咱们前面已经介绍过这个宏了。

在管道里面，第 7 位 (bit 7) 是表示方向的。所以这里 0x80 也就是说设 bit 7 为 1，这就表示传输方向是由设备向主机的，也就是所谓的 IN；而如果这一位是 0，就表示传输方向是由主机向设备的，也就是所谓的 OUT。而正是因为 USB\_DIR\_OUT 是 0，而 USB\_DIR\_IN 是 1，所以我们看到定义管道时只有用到了 USB\_DIR\_IN，而没有用到 USB\_DIR\_OUT，因为它是 0，任何数和 0 相或都没有意义。

这样，咱们就知道了，get\_pipes 函数中 764 行，765 行就是为 us 的控制输入和控制输出管道赋了值，管道是单向的。但是有一个例外，那就是控制端点。控制端点是双向的，比如 36

号楼 201 这个端点既可以是往外寄东西，也可以是作为收件人地址。而 USB 规范规定了，每一个 USB 设备至少得有一个控制端点，其端点号为 0。其他端点有没有得看具体设备而定，但这个端点是不管是什么设备，只要在 USB 中，那你就得遵守这个规矩，没得商量。所以我们看到 764 行，765 行里传递的 `endpoint` 变量值为 0。显然其构造的两个管道就是对应这个 0 号控制端点的。而接下来几行，就是构造 `bulk` 管道和中断管道（如果有中断端点的话）。

对于批量端点和中断端点（如果有的话），在它们的端点描述符里有一个字段 `bEndpointAddress`，这个字段共八位，但是它包含了挺多信息的，比如这个端点是输入端点还是输出端点，比如这个端点的地址（总线枚举时给它分配的），以及这个端点的端点号。不过要取得它的端点号得用一个掩码 `USB_ENDPOINT_NUMBER_MASK`，让 `bEndpointAddress` 和 `USB_ENDPOINT_NUMBER_MASK` 相与就能得到它的端点号。（就好比一份藏头诗，你得按着特定的方法才能读懂它，而这里特定的方法就是和 `USB_ENDPOINT_NUMBER_MASK` 这个掩码相与就行了。）

773 行，对于中断端点，您还得使用端点描述符中的 `bInterval` 字段，表示端点的中断请求间隔时间。

至此，`get_pipes` 函数结束了，信息都保存到了 `us` 里面。下面 `us` 该发挥它的作用了。回到 `storage_probe()` 函数。1005 行，把 `us` 作为参数传递给了 `usb_stor_acquire_resources()` 函数。而这个函数才是故事的高潮。每一个有识之士在看过这个函数之后就会豁然开朗，都会感慨，一下子就看到了春天，看到了光明，原来 Linux 中的设备驱动程序就是这么工作的啊！

```

1004    /* Acquire all the other resources and add the host */
1005    result = usb_stor_acquire_resources(us);
1006    if (result)
1007        goto BadDevice;
1008    result = scsi_add_host(host, &intf->dev);
1009    if (result) {
1010        printk(KERN_WARNING USB_STORAGE
1011               "Unable to add the scsi host\n");
1012        goto BadDevice;
1013    }

```

我们来看 `usb_stor_acquire_resources` 函数。它被定义于 `drivers/usb/storage/usb.c` 中：

```

778 /* Initialize all the dynamic resources we need */
779 static int usb_stor_acquire_resources(struct us_data *us)
780 {
781     int p;
782     struct task_struct *th;
783
784     us->current_urb = usb_alloc_urb(0, GFP_KERNEL);
785     if (!us->current_urb) {
786         US_DEBUGP("URB allocation failed\n");
787         return -ENOMEM;
788     }
789
790     /* Just before we start our control thread, initialize

```

```

791     * the device if it needs initialization */
792     if (us->unusual_dev->initFunction) {
793         p = us->unusual_dev->initFunction(us);
794         if (p)
795             return p;
796     }
797
798     /* Start up our control thread */
799     th = kthread_create(usb_stor_control_thread, us, "usb-storage");
800     if (IS_ERR(th)) {
801         printk(KERN_WARNING USB_STORAGE
802             "Unable to start control thread\n");
803         return PTR_ERR(th);
804     }
805
806     /* Take a reference to the host for the control thread and
807      * count it among all the threads we have launched. Then
808      * start it up. */
809     scsi_host_get(us_to_host(us));
810     atomic_inc(&total_threads);
811     wake_up_process(th);
812
813     return 0;
814 }

```

“待到山花烂漫时，她在丛中笑”。一个悟性高的人应该一眼就能从这个函数中找出那行“在丛中笑”的代码来，没错，它就是 799 行，`kthread_create()`，这个函数造就了许多经典的 Linux 内核模块，正是因为它的存在，Linux 中某些设备驱动程序的编写变得非常简单。

可以说，对某些设备驱动程序来说，`kthread_create()`几乎是整个驱动的灵魂，或者说是该 Linux 内核模块的灵魂。不管它隐藏得多么深，她总像漆黑中的萤火虫，那样鲜明，那样出众。甚至不夸张地说，对于很多模块来说，只要找到 `kthread_create()`这一行，基本上你就知道这个模块是怎么工作的了。

## 23. 传说中的 URB

有人问，怎么写一个驱动写这么久啊？

的确，一路走来，大家都不容易，但既然已经走到今天，我们能做的也只有坚持下去。

`usb_stor_acquire_resources()`，从名字上来看是获取资源。什么是资源？之前不是申请了一大堆内存了吗？写个 USB 设备驱动程序怎么这么麻烦啊？不是专门为 USB Mass Storage 设备准备了一个 `struct us_data` 这么一个结构体了吗？不是说故事已经到高潮了吗？

如果你以为看到这里你已经对 USB 设备驱动程序有了足够的认识，认为接下来的代码已经没有必要再分析了，那么，我只想说，上帝创造世界的计划中，未必包括使你会写 USB 设备

驱动程序。

的确，别看 `usb_stor_acquire_resources` 的代码不多，每一行都有每一行的故事。本节我们只讲其中的一行代码，没错，就是一行代码，因为我们需要隆重推出一个名词，一个响当当的名字，它就是传说中的“urb”，全称 USB Request Block。USB 设备需要通信，要传递数据，就需要使用 urb，确切地说，应该是 USB 设备驱动程序使用 urb。实际上，作为 USB 设备驱动，它本身并不能直接操纵数据的传输，在 USB 这个大观园里，外接设备永远都是配角，真正的核心只是 USB Core，而真正负责调度的是 USB 主机控制。这个通常看不见的 USB 主机控制器芯片，俨然是 USB 大观园中的大管家。设备驱动要发送信息，所需要做的是建立一个 urb 数据结构，并把这个数据结构交给核心层，而核心层会为所有设备统一完成调度，而设备在提交了 urb 之后需要做的，只是等待。别急，我们慢慢来。

784 行，一条赋值语句，等号左边 `us->current_urb`，等号右边 `usb_alloc_urb()` 函数被调用。如果说 `struct us_data` 是 usb mass storage 中的主角，那么 `struct urb` 将毫无争议地成为整个 USB 子系统的主角。Linux 中所有的 USB 设备驱动，都必然也必须要使用 urb。那么 urb 究竟长成什么样呢？在 `include/linux/usb.h` 中能找到它：

```
1126 struct urb
1127 {
1128     /* private: usb core and host controller only fields in the urb */
1129     struct kref kref;                /* reference count of the URB */
1130     spinlock_t lock;                /* lock for the URB */
1131     void *hcpriv;                   /* private data for host controller */
1132     atomic_t use_count;              /* concurrent submissions counter */
1133     u8 reject;                       /* submissions will fail */
1134
1135     /* public: documented fields in the urb that can be used by drivers*/
1136     struct list_head urb_list;       /* list head for use by the urb's
1137                                         * current owner */
1138     struct usb_device *dev;          /* (in) pointer to associated device */
1139     unsigned int pipe;               /* (in) pipe information */
1140     int status;                      /* (return) non-ISO status */
1141     unsigned int transfer_flags;     /* (in) URB_SHORT_NOT_OK | ...*/
1142     void *transfer_buffer;           /* (in) associated data buffer */
1143     dma_addr_t transfer_dma;         /* (in) dma addr for transfer_buffer */
1144     int transfer_buffer_length;      /* (in) data buffer length */
1145     int actual_length;               /* (return) actual transfer length */
1146     unsigned char *setup_packet;     /* (in) setup packet (control only) */
1147     dma_addr_t setup_dma;            /* (in) dma addr for setup_packet */
1148     int start_frame;                 /* (modify) start frame (ISO) */
1149     int number_of_packets;           /* (in) number of ISO packets */
1150     int interval;                   /* (modify) transfer interval
1151                                         * (INT/ISO) */
1152     int error_count;                 /* (return) number of ISO errors */
1153     void *context;                   /* (in) context for completion */
1154     usb_complete_t complete;         /* (in) completion routine */
1155     struct usb_iso_packet_descriptor iso_frame_desc[0];
1156                                         /* (in) ISO ONLY */
1157 };
```

我们常常抱怨，Linux 内核源代码中注释太少了，以至于我们常常看不懂那些代码究竟是什么含义。但 `urb` 让开发人员做足了文章，结构体 `struct urb` 的定义不过 30 行，而说明文字却用了足足 160 余行。可见 `urb` 的地位。当然我们这里贴出来主要还是为了保持原汁原味，另一方面这个注释也说得很清楚，对于每一个成员都解释了，而我们接下来将必然要用到 `urb` 的许多成员。

此刻，我们暂时不去理会这个结构体每一个元素的作用，只需要知道，一个 `urb` 包含了执行 USB 传输所需要的所有信息。而作为驱动程序，要通信就必须创建这么一个数据结构，并且赋值，显然不同类型的传输，需要对 `urb` 赋不同的值，然后将她提交给底层，完了底层的 USB Core 会找到相应的 USB 主机控制器，从而具体实现数据的传输。传输完了之后，USB 主机控制器会通知设备驱动程序。

总之我们知道，784 行就是调用 `usb_alloc_urb()` 申请了一个 `struct urb` 结构体。关于 `usb_alloc_urb()` 这个函数，我们打算讲，它是 USB Core 所提供的一个函数，来自 `drivers/usb/core/urb.c`，USB 开发人员的确是给足了 `urb` 的面子，专门把和这个数据结构相关的代码整理在这么一个文件中了。我们可以在 `include/linux/usb.h` 中找到这个函数的声明：

```
1266 extern struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
```

这个函数的作用很明显，就是为一个 `urb` 结构体申请内存。它有两个参数，其中第一个 `iso_packets` 用来在等时传输的方式下指定你需要传输多少个包，对于非等时模式来说，这个参数直接使用 0。另一个参数 `mem_flags` 就是一个 `flag`，表示申请内存的方式，这个 `flag` 将最终传递给 `kmalloc` 函数，我们这里传递的是 `GFP_KERNEL`，这个 `flag` 是内存申请中最常用的，我们之前也用过，在为 `us` 申请内存时。`usb_alloc_urb` 最终将返回一个 `urb` 指针，而 `us` 的成员 `current_urb` 也是一个 `struct urb` 的指针，所以就赋给它了。不过需要记住，`usb_alloc_urb` 除了申请内存以外，还对结构体做了初始化，结构体 `urb` 被初始化为 0，虽然这里我们没有把这个函数的代码“贴”出来，但你也千万不要以为写代码的人跟我似的，申请变量还能忘了初始化。同时，`struct urb` 中还有一个引用计数，以及一个自旋锁，这些也同样被初始化了。

所以，接下来我们就将要和 `us->current_urb` 打交道了。如果你对 `urb` 究竟怎么用还有些困惑的话，可以查看主机控制器驱动的代码。如果你不想看，那么我可以用一种你最能接受的方式告诉你，USB 是一种总线，是总线它就要通信。我们现实生活中真正要使用的是设备，但是光有设备还不足以实现 USB 通信，于是世界上有了 USB 主机控制器，它来负责统一调度。这就好比城市的交警，这个城市里真正需要的本来是车辆和行人，而光有车辆和行人，没有交警，那么这个城市里的车辆和行人必将乱套。于是诞生了交警这个行业，交警站在路口统一来管理调度混乱的交通。假如车辆和行人可以完全自觉遵守某种规矩而来来往往于这个城市的每一个角落及每一个路口，那么交警就没有必要存在了。同样，假如设备能够完全自觉地传递信息，每一个数据包都能到达它应该去的地方，那么我们根本就不需要有主机控制器。然而，事实上总会有不遵守交通规则的人。同样，在 USB 的世界中，设备也总是那么不守规矩，我们必须要

设计一个东西出来管理来控制所有的 USB 设备的通信，这样，主机控制器就横空出世了。

那么设备和主机控制器的分工又是如何呢？硬件实现上我们就不说了，说点儿具体的，在 Linux 中，设备驱动程序只要为每一次请求准备一个 `urb` 结构体变量，把它填充好（就是说赋上该赋的值），然后它调用 USB Core 提供的函数，把这个 `urb` 传递给主机控制器，主机控制器就会把各个设备驱动程序所提交的 `urb` 统一规划，去执行每一个操作。而这期间，USB 设备驱动程序通常会进入睡眠，而一旦主机控制器把 `urb` 要做的事情给做完了，它会调用一个函数去唤醒 USB 设备驱动程序，然后 USB 设备驱动程序就可以继续往下走了。

这又好比我们学校里的师生关系。考试时，我们只管把试卷填好，然后我们交给老师，然后老师拿去批改试卷，这期间我们除了等待别无选择，等待老师改完了试卷，告诉了我们分数，我们又继续我们的生活。同样，USB 设备驱动程序也是如此，如果 `urb` 提交给 USB 主机了，但是最终却没有成功执行，那么也许该 USB 设备驱动程序的生命也就提前结束。不过这都是后话，现在只要有一个感性认识即可，稍后看到了就能更深刻的体会了，这种岗位分工的方式给我们编写设备驱动程序带来了巨大的方便。

继续 `usb_stor_acquire_resources` 函数。

785 行到 788 行，就是刚才 `urb` 申请了之后判断是否申请成功了，如果指针为 `NULL` 那么就是失败了，直接返回-`ENOMEM`。

792 行，`us->unusual_dev->initFunction` 是什么？在分析 `unusual_devs.h` 文件时曾经专门举过例子的，说有些设备需要一些初始化函数，它就定义在 `unusual_devs.h` 文件中，而我们通过 `UNUSUAL_DEV` 的定义已经把这些初始化函数赋给了 `us->unusual_dev` 的 `initFunction` 指针了。所以这时候，在传输开始之前，我们判断是不是有这样一个函数，即这个函数指针是否为空，如果不为空，很好办，执行这个函数就行了。比如当时我们举例子时说的那两个设备就有初始化函数，那么就让它执行好了。当然，一般的设备肯定不需要这么一个函数。至于传递给这个函数的参数，在 `struct us_unusual_dev` 结构体定义时，就把这个函数需要什么样的参数定义好了，需要的就是一个 `struct us_data *`，那么很自然，传递的就是 `us`。

至此，我们终于走到了 `usb_stor_acquire_resources()` 中第 799 行，即将见到这个千呼万唤始出来的内核精灵。

---

## 24. 彼岸花的传说（一）

彼岸花，花语是悲伤的回忆。

很久很久以前，城市的边缘开满了大片大片的曼珠沙华，它的花香有一种魔力，可以让人想起自己前世的事情。守护曼珠沙华的是两个妖精，一个是花妖叫曼珠，一个是叶妖叫沙华。他们守候了几千年，可是从来没有见过面，因为开花时，就没有叶子，有叶子时没有花。他们疯狂地想念着彼此，并被这种痛苦折磨着。终于有一天，他们决定违背神的规定偷偷地见一次面。那一年的曼珠沙华红艳艳的花被惹眼的绿色衬托着，开得格外妖冶美丽。

曼珠和沙华受到惩罚，被打入轮回，并被诅咒永远也不能在一起，生生世世在人世间受到磨难。从那以后，曼珠沙华又叫彼岸花，意思是开放在天国的花，它的花的形状像一只只在向天堂祈祷的手掌，可是再也没有在这个城市出现过。每年的秋彼岸期间（春分前后三天叫春彼岸，秋分前后三天叫秋彼岸）她会开在黄泉路上，曼珠和沙华的每一次转世在黄泉路上闻到彼岸花的香味就能想起前世的自己，然后发誓不分开，但只有在这一刻，因为他们会再次跌入诅咒的轮回，灵魂借着花的指引，走向幽冥。

Linux 内核中引入了守护进程，也正是与这个传说对应，守护进程也叫内核精灵，当然，如果你是无神论者，你可以叫它为内核线程。我们来看具体的代码。

799 行，调用了 `kthread_create()` 函数，`kthread_create(usb_stor_control_thread, us, "usb-storage")`，如果从前您对内核本身不是很熟悉，那这个函数就会让你有点儿头疼了。这个函数将会创建一个内核线程，而函数 `usb_stor_control_thread()` 将会执行，`us` 将是传递给它的参数，对 Linux 内核不是很熟悉的话，可以将 `kthread_create` 看做类似于 `fork` 的函数。

实际上，简单一点说，`kthread_create()` 这么一执行呢，就会有两个进程，一个是父进程，一个是子进程，子进程将会执行 `usb_stor_control_thread()`，而 `us` 是作为 `usb_stor_control_thread` 函数的参数（实参），执行完 `usb_stor_control_thread()` 之后，子进程就结束了，它会调用 `exit()` 函数退出。而父进程继续顺着 `usb_stor_acquire_resources()` 函数往下走，`kthread_create()` 函数对于父进程而言返回的是子进程的进程 `task_struct` 结构体指针，800 行调用 `IS_ERR(th)` 判断返回的指针是否是错误代码，若是 `IS_ERR(th)` 为真，则调用 `PTR_ERR(th)` 读出实际的错误代码。

于是，咱们接下来必须再次兵分两路，分别跟踪父进程和子进程前进了。先看父进程，811 行：

```
811      wake_up_process(th);
```

唤醒子进程，之所以需要唤醒子进程，是因为当你用 `kthread_create()` 创建一个子进程之后，它并不会立即执行，它要等待你唤醒了之后才会执行，所以这个函数就相当于田径运动中裁判的发令声。运动员跑得再快，他也要等到裁判发令声之后才会开始跑。那么我们来看子进程，也就是 `usb_stor_control_thread()` 函数，这个函数定义于 `drivers/usb/storage/usb.c` 中。

## 25. 彼岸花的传说（二）

如果大家投票的话，usb\_stor\_control\_thread()这个函数中的代码无疑是整个模块中最为精华的代码。我们只需要它中间 306 行的 for(;;)就知道，这是一个死循环，即使别的代码都执行完了，即使别的函数都退出了，这个函数也仍然像永不消逝的电波一般，经典常驻。显然，只有死循环才能代表永恒，才能代表忠诚。这是每一个守护者的职责。

usb\_stor\_control\_thread()，其代码如下：

```

299 static int usb_stor_control_thread(void * __us)
300 {
301     struct us_data *us = (struct us_data *)__us;
302     struct Scsi_Host *host = us_to_host(us);
303
304     current->flags |= PF_NOFREEZE;
305
306     for(;;) {
307         US_DEBUGP("*** thread sleeping.\n");
308         if(down_interruptible(&us->sema))
309             break;
310
311         US_DEBUGP("*** thread awakened.\n");
312
313         /* lock the device pointers */
314         mutex_lock(&(us->dev_mutex));
315
316         /* if the device has disconnected, we are free to exit */
317         if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
318             US_DEBUGP("-- exiting\n");
319             mutex_unlock(&(us->dev_mutex));
320             break;
321         }
322
323         /* lock access to the state */
324         scsi_lock(host);
325
326         /* has the command timed out *already* ? */
327         if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
328             us->srb->result = DID_ABORT << 16;
329             goto SkipForAbort;
330         }
331
332         scsi_unlock(host);
333
334         /* reject the command if the direction indicator
335          * is UNKNOWN
336          */
337         if (us->srb->sc_data_direction == DMA_BIDIRECTIONAL) {
338             US_DEBUGP("UNKNOWN data direction\n");
339             us->srb->result = DID_ERROR << 16;
340         }
341
342         /* reject if target != 0 or if LUN is higher than
343          * the maximum known LUN

```



```

344     */
345     else if (us->srb->device->id &&
346             !(us->flags & US_FL_SCM_MULT_TARG)) {
347         US_DEBUGP("Bad target number (%d:%d)\n",
348                 us->srb->device->id, us->srb->device->lun);
349         us->srb->result = DID_BAD_TARGET << 16;
350     }
351
352     else if (us->srb->device->lun > us->max_lun) {
353         US_DEBUGP("Bad LUN (%d:%d)\n",
354                 us->srb->device->id, us->srb->device->lun);
355         us->srb->result = DID_BAD_TARGET << 16;
356     }
357
358     /* Handle those devices which need us to fake
359     * their inquiry data */
360     else if ((us->srb->cmdnd[0] == INQUIRY) &&
361             (us->flags & US_FL_FIX_INQUIRY)) {
362         unsigned char data_ptr[36] = {
363             0x00, 0x80, 0x02, 0x02,
364             0x1F, 0x00, 0x00, 0x00};
365
366         US_DEBUGP("Faking INQUIRY command\n");
367         fill_inquiry_response(us, data_ptr, 36);
368         us->srb->result = SAM_STAT_GOOD;
369     }
370
371     /* we've got a command, let's do it! */
372     else {
373         US_DEBUG(usb_stor_show_command(us->srb));
374         us->proto_handler(us->srb, us);
375     }
376
377     /* lock access to the state */
378     scsi_lock(host);
379
380     /* did the command already complete because of a disconnect? */
381     if (!us->srb)
382         ; /* nothing to do */
383
384     /* indicate that the command is done */
385     else if (us->srb->result != DID_ABORT << 16) {
386         US_DEBUGP("scsi cmd done, result=0x%x\n",
387                 us->srb->result);
388         us->srb->scsi_done(us->srb);
389     } else {
390     SkipForAbort:
391         US_DEBUGP("scsi command aborted\n");
392     }
393
394     /* If an abort request was received we need to signal that
395     * the abort has finished. The proper test for this is
396     * the TIMED_OUT flag, not srb->result == DID_ABORT, because
397     * the timeout might have occurred after the command had
398     * already completed with a different result code. */
399     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
400         complete(&(us->notify));
401
402         /* Allow USB transfers to resume */

```

```

403         clear_bit(US_FLIDX_ABORTING, &us->flags);
404         clear_bit(US_FLIDX_TIMED_OUT, &us->flags);
405     }
406
407     /* finished working on this command */
408     us->srb = NULL;
409     scsi_unlock(host);
410
411     /* unlock the device pointers */
412     mutex_unlock(&us->dev_mutex);
413 } /* for (;;) */
414
415     scsi_host_put(host);
416
417     /* notify the exit routine that we're actually exiting now
418     *
419     * complete()/wait_for_completion() is similar to up()/down(),
420     * except that complete() is safe in the case where the structure
421     * is getting deleted in a parallel mode of execution (i.e. just
422     * after the down() -- that's necessary for the thread-shutdown
423     * case.
424     *
425     * complete_and_exit() goes even further than this -- it is safe in
426     * the case that the thread of the caller is going away (not just
427     * the structure) -- this is necessary for the module-remove case.
428     * This is important in preemption kernels, which transfer the flow
429     * of execution immediately upon a complete().
430     */
431     complete_and_exit(&threads_gone, 0);
432 }

```

302 行,定义了一个 Scsi\_Host 的指针 host,令它指向 us->host,也就是刚刚用 scsi\_host\_alloc() 申请的 Scsi\_Host 结构体变量。

304 行,这里为目前的进程设置一个 flag, PF\_NOFREEZE, 在整个内核代码中,这个 flag 也只出现过几次。这个 flag 是与电源管理相关的,2.6 内核为了实现与 Windows 相似的一个功能——Hibernate,也就是“冬眠”,(Windows 关机选项里面有“关机”,“重启”,“注销”,“Stand by”,以及“Hibernate”)。在内核编译菜单里面,Power managerment options 中,有一个选项 Software Suspend,也就是内核编译选项中的 CONFIG\_SOFTWARE\_SUSPEND,选择了它使得机器可以被 suspended (挂起)。显然咱们不用在意它。但是这里之所以要设置这个 flag,是因为 suspend 要求把内存里的内容写到磁盘上,而一个进程设置了这个 flag 就表明它在 suspend 时不会被冻住,用“行话”来讲就是“they’re not refrigerated during a suspend.” freeze 就是冷冻,冻住的意思,过去分词 frozen 是形容词,冻结的,冰冻的,其实就是让进程睡眠。所以总的来说,即使系统“suspend”了,这个进程或者准确地说这个内核线程,也不应该进入睡眠。

306 行,一个 for 语句死循环,尽管外面的世界很精彩,但是咱们不妨去查看 for 里面的世界。

308 行,down\_interruptible()函数,事实上 307 行的调式信息已经告诉我们,thread 将进入睡眠了。down\_interruptible 的参数是&us->sema,这是一把锁,而这里就是想获得这把锁,但

是别忘了，这把锁一开始就被初始化为 0 了，参考 `drivers/usb/storage/usb.c` 中第 973 行，`storage_probe()` 函数：

```
973         init_MUTEX_LOCKED(&(us->sema));
```

也就是说它属于那种“指腹为婚”的情形，一到这个世界来就告诉别人自己已经是名花有主了。因此，这里只能进入睡眠，等待一个 `up()` 函数去释放锁。谁会调用 `up()` 函数呢？暂时先不管它，我们先关注一下父进程，父进程在执行完 `wake_up_process(th)` 之后，`usb_stor_acquire_resources()` 将结束，带着 0 返回了 `storage_probe()` 中去。

## 26. 彼岸花的传说（三）

前面已经说了，回到 `usb_stor_acquire_resources()` 函数中，返回了 0。于是咱们终于回到了 `storage_probe()` 函数中来。

1008 行，`scsi_add_host()` 函数被执行，之前申请的 `us->host` 被作为参数传递给它，同时，`intf->dev` 也被传递给它，这个东西是被用来注册 `sysfs` 的。前面已经说过，在 `scsi_host_alloc` 之后，必须执行 `scsi_add_host()`，这样，SCSI 核心层才能够知道有这么一个 `host` 存在。`scsi_add_host()` 成功则返回 0，否则返回出错代码。如果一切顺利，将走到 1009 行，别急，先把代码“贴”出来，这就是 `storage_probe()` 函数的最后一小段了：

```
1014
1015     /* Start up the thread for delayed SCSI-device scanning */
1016     th = kthread_create(usb_stor_scan_thread, us, "usb-stor-scan");
1017     if (IS_ERR(th)) {
1018         printk(KERN_WARNING USB_STORAGE
1019                "Unable to start the device-scanning thread\n");
1020         quiesce_and_remove_host(us);
1021         result = PTR_ERR(th);
1022         goto BadDevice;
1023     }
1024
1025     /* Take a reference to the host for the scanning thread and
1026      * count it among all the threads we have launched. Then
1027      * start it up. */
1028     scsi_host_get(us_to_host(us));
1029     atomic_inc(&total_threads);
1030     wake_up_process(th);
1031
1032     return 0;
1033
1034     /* We come here if there are any problem ms */
1035 BadDevice:
1036     US_DEBUGP("storage_probe() failed\n");
1037     release_everything(us);
1038     return result;
```

1039 }

又一次见到了 `kthread_create`，不需要更多解释，这里自然还是创建一个内核守护进程，只不过这次是 `usb_stor_scan_thread`，而上次是 `usb_stor_control_thread`。`usb_stor_scan_thread()` 函数也是定义于 `drivers/usb/storage/usb.c` 中：

```

904 /* Thread to carry out delayed SCSI-device scanning */
905 static int usb_stor_scan_thread(void * __us)
906 {
907     struct us_data *us = (struct us_data *)__us;
908
909     printk(KERN_DEBUG
910            "usb-storage: device found at %d\n", us->pusb_dev->devnum);
911
912     /* Wait for the timeout to expire or for a disconnect */
913     if (delay_use > 0) {
914         printk(KERN_DEBUG "usb-storage: waiting for device "
915            "to settle before scanning\n");
916     retry:
917         wait_event_interruptible_timeout(us->delay_wait,
918            test_bit(US_FLIDX_DISCONNECTING, &us->flags),
919            delay_use * HZ);
920         if (try_to_freeze())
921             goto retry;
922     }
923
924     /* If the device is still connected, perform the scanning */
925     if (!test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
926
927         /* For bulk-only devices, determine the max LUN value */
928         if (us->protocol == US_PR_BULK &&
929            !(us->flags & US_FL_SINGLE_LUN)) {
930             mutex_lock(&us->dev_mutex);
931             us->max_lun = usb_stor_Bulk_max_lun(us);
932             mutex_unlock(&us->dev_mutex);
933         }
934         scsi_scan_host(us_to_host(us));
935         printk(KERN_DEBUG "usb-storage: device scan complete\n");
936
937         /* Should we unbind if no devices were detected? */
938     }
939
940     scsi_host_put(us_to_host(us));
941     complete_and_exit(&threads_gone, 0);
942 }

```

913 行，`delay_use` 哪来的？同一个文件中，最开始的地方，定义了一个静态变量：

```

110 static unsigned int delay_use = 5;
111 module_param(delay_use, uint, S_IRUGO | S_IWUSR);
112 MODULE_PARM_DESC(delay_use, "seconds to delay before using a new device");

```

设置了 `delay_use` 为 5，而 `module_param` 是 Linux Kernel 2.6 提供的一个宏，使得 `delay_use` 可以在模块被装载时设定。（如果不设，那么它自然就是这里的值 5，表示使用一个新的设备之前等待 5 秒延时。）为什么要延时啊？当插进去的 U 盘也可能立刻又被拔出来了，试想插入以

后一两秒之内又拔出来，那么咱们下面也不用耽误工夫再检测了。

913 行，判断 `delay_use>0`，然后 917 行，`wait_event_interruptible_timeout()`，它的第一个参数是 `us->delay_wait`。

在 `storage_probe()` 函数的最初，在 `us` 的初始化时，`delay_wait` 被初始化了。975 行，`init_waitqueue_head(&us->delay_wait)`，而在定义 `struct us_data` 时，有一个成员就是 `delay_wait`，即 `wait_queue_head_t delay_wait`，这些都是什么意思呢？

实际上 `wait_event_interruptible_timeout()` 是一个宏，它代表着 Linux 中的一种等待机制，等待某个事件的发生，函数原型中，第 1 个参数是一个等待队列头，即 `wait_queue_head_t` 定义的变量，在 2.6 内核中使用 `init_waitqueue_head()` 函数初始化这个等待队列，然后第 3 个参数是设置超时。比如这里设了 5 秒，这表示如果 5 秒到了，那么函数会返回 0，不管其他条件如何。第 2 个参数是一种等待的条件，或者说等待的事件，如果条件满足了，那么函数也会返回，条件要是不满足，那么这个进程会进入睡眠，不过 `interruptible` 表明了信号可以把它中断。

一旦进入睡眠，那么有三种情况：一种是 `wake_up` 或者 `wake_up_interruptible` 函数被另一个进程执行，从而唤醒它，第二种是信号中断它，第三种就是刚才讲的超时，时间到了，自然就会返回。

那么这里具体来说，先判断 `US_FLIDX_DISCONNECTING` 这个 flag 有没有设置，如果没有设置才进入睡眠，否则就不需要浪费彼此的感情了。在进入睡眠之后，如果 5 秒之内没有把 U 盘拔出来，那么 5 秒一到，函数返回 0，继续往下走，如果在 5 秒之前拔出来 U 盘了，那么后来咱们会讲，`storage_disconnect()` 函数会执行，它会设置 `US_FLIDX_DISCONNECTING` 这个 flag，并且它会调用 `wake_up(&us->scsi_scan_wait)` 来唤醒这里睡眠的进程，告诉它：“别等了，哥们儿，你没那种命！”这样函数就会提前返回，不用等到 5 秒再返回了。总之不管条件满不满足，5 秒之内肯定会返回，所以我们继续往下看。

920 行，`try_to_freeze()`，这是电源管理的内容。

925 行，再次判断设备有没有被断开，如果还是没有，那么执行 `scsi_scan_host()` 函数扫描，扫描然后就知道这个 `host` 或者说这个 SCSI 卡上面接了什么设备（虽然咱们这个只是模拟的 SCSI 卡），然后 `cat /proc/scsi/scsi` 才能看到您的 U 盘。

928 行这一段，就是对于有多个 LUN 的设备，调用 `usb_stor_Bulk_max_lun()` 来获得 `max_lun`。

然后 941 行，`complete_and_exit` 函数，它和 `complete` 函数还有一点不一样，除了唤醒别人，还得结束自己（`exit`）。它在 `kernel/exit.c` 中：

```
1010 NORET_TYPE void complete_and_exit(struct completion *comp, long code)
1011 {
1012     if (comp)
1013         complete(comp);
```

```

1014
1015     do_exit(code);
1016 }

```

这个函数中最重要的是 `do_exit()` 函数，不用多说，它是内核提供的函数，结束进程。也就是说，对于上面这个 `scan` 的精灵进程，到这里它就会结束退出了。可以看出它是一个短命的守护进程。总之对于这个精灵进程来说，它的使命就是让你能在 `cat /proc/scsi/scsi` 中看到你的 U 盘，当然了，从此以后你在 `/dev` 目录下面也就能看到你的设备了，比如 `/dev/sda`。

再来看父进程，也就是 `storage_probe()`，在用 `kernel_thread()` 创建了 `usb_stor_scan_thread` 之后，一切正常的话，`storage_probe()` 也走到了尽头了。1032 行，`return 0` 了。终于，这个不老的传说也终于到了老的那一刻，一切都结束了，一切都烟消云散了。

## 27. 彼岸花的传说（四）

我们刚刚跟着 `storage_probe()` 几乎完整地走了一遍，貌似一切都该结束了，可是你不觉得你到目前为止还根本没有看明白设备究竟怎么工作的吗？U 盘，不仅仅是 USB 设备，还是“盘”，它还需遵守 USB Mass Storage 协议，以及 Transparent SCSI 规范。从驱动程序的角度来看，它和一般的 SCSI 磁盘差不多。正是因为如此，所以 U 盘的工作真正需要的是四个模块，`usbcore`，`scsi_mod`，`sd_mod`，以及咱们这里的 `usb-storage`，其中 `sd_mod` 恰恰就是 SCSI 硬盘的驱动程序。没有它，你的 SCSI 硬盘就别想在 Linux 下面转起来。

那么我们从哪里开始去接触这些 SCSI 命令呢？别忘了我们现在的主题，内核守护进程，别忘了我们曾经有一段代码只讲到一半就没讲了。没错，那就是 `usb_stor_control_thread()`，当初我们用 `kthread_create` 创建它时就说了，从此以后一个进程变成两个进程。而我们刚才沿着 `storage_probe` 讲完的是父进程，父进程最终返回了，而子进程则没有那么简单，我们已经说过，`usb_stor_control_thread()` 中的死循环注定了这个子进程是一个永恒的进程，只要这个模块还没有被卸载，或者说还没有被要求卸载，这个子进程就必将永垂不朽地战斗下去。于是让我们推开记忆的门，回过来看这个函数，当初我们讲到了 308 行，由于 `us->sema` 一开始就是锁着的，所以 `down_interruptible` 这里一开始就进入睡眠了，只有在接到唤醒的信号或者锁被释放了释放锁的进程来唤醒它，它才会醒过来。那么谁来释放这把锁呢？

有两个地方，一个是这个模块要卸载了，这个我们稍后来看。另一个就是有 SCSI 命令发过来了。SCSI 命令从哪里发过来？很简单，SCSI 核心层，硬件上来说，SCSI 命令就是 SCSI 主机到 SCSI 设备，而从代码的角度来说，SCSI 核心层要求为每一个 SCSI 主机实现一个 `queuecommand` 命令，每一次应用层发送来了 SCSI 命令了，比如你去读写 `/dev/sda`，最终 SCSI 核心层就会调用与该主机相对应 `queuecommand`，（确切地说是 `struct Scsi_Host` 结构体中的成员

struct scsi\_host\_template 中的成员指针 queuecommand，这是一个函数指针。) 那么我们来看，当初我们定义的 struct scsi\_host\_template usb\_stor\_host\_template，其中的确有一个 queuecommand，我们赋给它的值也是 queuecommand，即我们让 queuecommand 指向一个叫做 queuecommand 的函数，在 struct scsi\_host\_template 的定义中，函数指针的原型来自 include/scsi/scsi\_host.h:

```
124     int (* queuecommand)(struct scsi_cmnd *,
125                          void (*done)(struct scsi_cmnd *));
```

而我们所定义的 queuecommand() 函数又在哪里呢？在 drivers/usb/storage/scsiglue.c 中：

```
208 /* queue a command */
209 /* This is always called with scsi_lock(host) held */
210 static int queuecommand(struct scsi_cmnd *srb,
211                        void (*done)(struct scsi_cmnd *))
212 {
213     struct us_data *us = host_to_us(srb->device->host);
214
215     US_DEBUGP("%s called\n", __FUNCTION__);
216
217     /* check for state-transition errors */
218     if (us->srb != NULL) {
219         printk(KERN_ERR USB_STORAGE "Error in %s: us->srb = %p\n",
220                __FUNCTION__, us->srb);
221         return SCSI_MLQUEUE_HOST_BUSY;
222     }
223
224     /* fail the command if we are disconnecting */
225     if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
226         US_DEBUGP("Fail command during disconnect\n");
227         srb->result = DID_NO_CONNECT << 16;
228         done(srb);
229         return 0;
230     }
231
232     /* enqueue the command and wake up the control thread */
233     srb->scsi_done = done;
234     us->srb = srb;
235     up(&(us->sema));
236
237     return 0;
238 }
```

这个函数不长，它的使命也很简单，就是为了唤醒那个沉睡中的守护进程，告诉它不能再沉睡。

我们来仔细看一下，213 行，us 的身影无处不在。

218 行，判断 us->srb，事到如今，我们不得不去面对一个新的数据结构，它就是 struct scsi\_cmnd。queuecommand() 函数的第 1 个参数就是 struct scsi\_cmnd 指针，而 struct us\_data 中也有一个 struct scsi\_cmnd \*srb，也是一个指针。

那么我们来看 struct scsi\_cmnd，这个数据结构的意义很明显，就是代表一个 SCSI 命令。

它定义于 `include/scsi/scsi_cmnd.h` 中。如果你感兴趣可以去看一下，我们只要知道有这么一个数据结构就可以了，同时需要知道在 `us` 中有一个成员 `srb`，由它来指向 `scsi` 命令。

继续说 218 行，看一下 `us->srb` 是不是为空，如果为空我们才可以继续往下走去唤醒那个守护进程，否则就说明之前的一个命令还没有执行完。它会执行完一个命令就会把它设为空。显然 `us->srb` 为空，因为还没有任何人为它赋过值，只是初始化 `us` 时把所有元素都初始化为 0 了，所以这第一次来到这里时肯定是空。这里如果不为空就返回 `SCSI_MLQUEUE_HOST_BUSY` 给 SCSI Core，这样核心层就知道，这边主机正忙着呢，先不急着执行下面的命令。

225 行，`US_FLIDX_DISCONNECTING` 这个 flag 我们已经不是第一次遇见了，无须多讲，现在只是不知道究竟是哪设置了这个 flag，日后我们看到 `storage_disconnect` 就知道了。这里和以往一样，如果这个 flag 设置了，赶紧结束吧。设置 `srb->result` 让 `scsi core` 知道这里已经断开连接了。而 `queuecommand` 命令本身就返回 0。不过我们需要注意的是 228 行这个 `done` 函数，仔细看这个 `done` 是 `queuecommand()` 函数的第 2 个参数，是一个函数指针，实际上 `scsi core` 调用 `queuecommand` 时传递的参数名字就叫做 `scsi_done`，这就是一个函数名，SCSI 核心层定义了一个叫做 `scsi_done` 的函数。SCSI 核心层要求当低层的驱动程序完成一个命令后要调用这个函数去通知 SCSI 核心层，这实际上就相当于一种中断机制。`scsi` 核心层调用了 `queuecommand()` 之后它就不管事了，它就去干别的了，等底层的代码把这个 `queuecommand` 执行完了之后，或者准确地说当底层把命令执行完了之后，就调用 `scsi_done` 从而 `scsi` 核心层就知道这个命令完成了，然后它就会接着做一些它该做的事情比如清理这个命令，或者别的一些收尾的工作。

所以这里我们看到，如果设备已经设置了断开的 flag，那么这里就执行 `done`，如果没有断开那就在下面的 233 行设置 `srb->scsi_done` 等于这个 `done`，实际上就是等于 `scsi_done`，这两个 `scsi_done`，一个是 `struct scsi_cmnd *srb` 的成员指针，一个是 SCSI 核心层的函数名。虽然它们同名，但是是两个不同的东西。

最后，234 行，令 `us->srb` 等于这个 `srb`。而 235 行，这正是我们苦苦寻找的代码，正是这个 `up(&us->sema)`，唤醒了我们的守护进程。之后，237 行，这个函数本身就结束了。而我们显然就该去看那个 `usb_stor_control_thread()` 了。因为，它醒了，它终于醒了。

## 28. 彼岸花的传说（五）

下面讲一下 `usb_stor_control_thread()` 函数。唤醒它的是来自 `queuecommand` 的 `up(&(us->sema))`，`us->srb` 被赋值为 `srb`，而 `srb` 是来自 SCSI 核心层在调用 `queuecommand` 时候传递进来的参数。聚焦 `usb_stor_control_thread()`。314 行，前面说过，关于 `dev_mutex` 这把锁我们必须在看完整个模块之后再从较高的角度来看。



312 行，如果设了 `US_FLIDX_DISCONNECTING`，这个不用多说了，是判断设备有没有被拔出，要是你的 U 盘插进去了永远不拔出来，那么你可以把这个 `flag` 相关的代码都删了，当然事实上是你不可能不拔出来，热插拔本来就是 USB 设备的一大特性。

324 行，`host` 也是一把锁，这把锁我们也到最后再来看。

326 行到 330 行，又是判断另一个 `flag` 有没有被设置，`US_FLIDX_TIMED_OUT` 这个 `flag` 的含义也如其字面意义一样，即超时了。超时的概念在计算机的世界里比比皆是。不过对于这个 `flag`，设置它的函数是 `command_abort`，这个函数也是咱们提供的，由 SCSI 核心层去调用，由它那边负责计时，到了超时的时间它就调用 `command_abort`。我们稍后会看，先不急。

337 行，判断 `srb` 的一个成员 `sc_data_direction`，先看 `DMA_BIDIRECTIONAL` 这个宏。这个宏定义于 `include/linux/dma-mapping.h` 中：

```
7 /* These definitions mirror those in pci.h, so they can be used
8  * interchangeably with their PCI_ counterparts */
9 enum dma_data_direction {
10     DMA_BIDIRECTIONAL = 0,
11     DMA_TO_DEVICE = 1,
12     DMA_FROM_DEVICE = 2,
13     DMA_NONE = 3,
14 };
```

这些代码被用来表示数据阶段数据传输的方向。`DMA_TO_DEVICE` 表示从主存到设备，`DMA_FROM_DEVICE` 表示从设备到主存。有传闻说，`DMA_NONE` 则只被用于调试，一般不能使用否则将有可能导致内核崩溃。不过更准确一点的是，USB Mass Storage 协议中边规定了双向传输是非法的，而一个命令传输零数据是合法的，比如 `TEST_UNIT_READY` 命令就不用传输数据。

`DMA_BIDIRECTIONAL` 表示两个方向都有可能，换言之也就是不知道究竟是哪个方向。同理，338 行看到 `srb` 的 `sc_data_direction` 是 `DMA_BIDIRECTIONAL` 时，自然就当做出错了。因为不确定方向的话也就没法传输数据了。

345 行，`US_FL_SCM_MULT_TARG` 这个 `flag`，表示设备支持多个 `target`，这里的意思很明显，对于那些不支持多个 `target` 的设备，其 `us->srb->device->id` 必须为 0，否则就有问题了。`struct us_data` 结构体中的成员 `struct scsi_cmnd * srb`，`struct scsi_cmnd` 结构体中有一成员 `struct scsi_device * device`，而 `struct scsi_device` 顾名思义，描述一个 SCSI 设备，就像过去的 `struct usb_device` 用来描述 USB 设备一样。`struct scsi_device` 来自 `include/scsi/scsi_device.h` 中：

```
49 struct scsi_device {
50     struct Scsi_Host *host;
51     struct request_queue *request_queue;
52
53     /* the next two are protected by the host->host_lock */
54     struct list_head siblings; /* list of all devices on this host */
55     struct list_head same_target_siblings;
```

```

56
57 /* this is now protected by the request_queue->queue_lock */
58 unsigned int device_busy; /* commands actually active on
59                          * low-level. protected by queue_lock. */
60 spinlock_t list_lock;
61 struct list_head cmd_list; /* queue of in use SCSI Command structures*/
62 struct list_head starved_entry;
63 struct scsi_cmnd *current_cmnd; /* currently active command */
64 unsigned short queue_depth; /* How deep of a queue we want */
65 unsigned short last_queue_full_depth; /* These two are used by */
66 unsigned short last_queue_full_count; /* scsi_track_queue_full() */
67 unsigned long last_queue_full_time; /*don't let QUEUE_FULLs on the same
68                                jiffie count on our counter, they
69                                could all be from the same event. */
70
71 unsigned int id, lun, channel;
72
73 unsigned int manufacturer; /* Manufacturer of device, for using
74                          * vendor-specific cmd's */
75 unsigned sector_size; /* size in Bytes */
76
77 void *hostdata; /* available to low-level driver */
78 char type;
79 char scsi_level;
80 char inq_periph_qual; /* PQ from INQUIRY data */
81 unsigned char inquiry_len; /* valid Bytes in 'inquiry' */
82 unsigned char * inquiry; /* INQUIRY response data */
83 const char * vendor; /* [back_compat] point into 'inquiry' ... */
84 const char * model; /* ... after scan; point to static string */
85 const char * rev; /* ... "nullnullnullnull" before scan */
86 unsigned char current_tag; /* current tag */
87 struct scsi_target *sdev_target; /* used only for single_lun*/
88
89 unsigned int sdev_bflags; /* black/white flags as also found in
90                          * scsi_devinfo.[hc]. For now used only to
91                          * pass settings from slave_alloc to scsi
92                          * core. */
93
94 unsigned writeable:1;
95 unsigned removable:1;
96 unsigned changed:1; /* Data invalid due to media change */
97 unsigned busy:1; /* Used to prevent races */
98 unsigned lockable:1; /* Able to prevent media removal */
99 unsigned locked:1; /* Media removal disabled */
100 unsigned borken:1; /* Tell the Seagate driver to be
101                   * painfully slow on this device */
102 unsigned disconnect:1; /* can disconnect */
103 unsigned soft_reset:1; /* Uses soft reset option */
104 unsigned sdtr:1; /* Device supports SDTR messages */
105 unsigned wdtr:1; /* Device supports WDTR messages */
106 unsigned ppr:1; /* Device supports PPR messages */
107 unsigned tagged_supported:1; /* Supports SCSI-II tagged queuing */
108 unsigned simple_tags:1; /* simple queue tag messages are enabled*/
109 unsigned ordered_tags:1; /* ordered queue tag messages are enabled */
110 unsigned single_lun:1; /* Indicates we should only allow I/O to
111                       * one of the luns for the device at a
112                       * time. */
113 unsigned was_reset:1; /* There was a bus reset on the bus for
114                       * this device */
115 unsigned expecting_cc_ua:1; /* Expecting a CHECK_CONDITION/UNIT_ATTN

```

```

115                                     * because we did a bus reset. */
116 unsigned use_10_for_rw:1; /* first try 10-Byte read / write */
117 unsigned use_10_for_ms:1; /* first try 10-Byte mode sense/select*/
118 unsigned skip_ms_page_8:1; /* do not use MODE SENSE page 0x08*/
119 unsigned skip_ms_page_3f:1; /* do not use MODE SENSE page 0x3f */
120 unsigned use_192_Bytes_for_3f:1; /* ask for 192 Bytes from page 0x3f*/
121 unsigned no_start_on_add:1; /* do not issue start on add */
122 unsigned allow_restart:1; /* issue START_UNIT in error handler */
123 unsigned manage_start_stop:1; /* Let HLD (sd) manage start/stop */
124 unsigned no_uld_attach:1; /* disable connecting to upper level drivers*/
125 unsigned select_no_atn:1;
126 unsigned fix_capacity:1; /* READ_CAPACITY is too high by 1 */
127 unsigned guess_capacity:1; /* READ_CAPACITY might be too high by 1 */
128 unsigned retry_hwerror:1; /* Retry HARDWARE_ERROR */
129
130 unsigned int device_blocked; /* Device returned QUEUE_FULL. */
131
132 unsigned int max_device_blocked;
133 #define SCSI_DEFAULT_DEVICE_BLOCKED 3
134
135 atomic_t iorequest_cnt;
136 atomic_t iodone_cnt;
137 atomic_t ioerr_cnt;
138
139 int timeout;
140
141 struct device sdev_gendev;
142 struct class_device sdev_classdev;
143
144 struct execute_work ew; /* used to get process context on put*/
145
146 enum scsi_device_state sdev_state;
147 unsigned long sdev_data[0];
148 } __attribute__((aligned(sizeof(unsigned long))));

```

这个结构体将在后面多次被提到。当然，此刻，我们只需要注意到 `unsigned int id`, `lun`, `channel` 这三个成员，这正是定位一个 SCSI 设备必要的三个成员，一个 SCSI 卡所控制的设备被划分为几层，先是若干个 `channel`，然后每个 `channel` 上有若干个 `target`，每个 `target` 用一个 `target id` 来表示，然后一个 `target` 可以有若干个 `lun`，而这里判断的是 `target id`。对于不支持多个 `target` 的设备，必须为 0。对于绝大多数 USB Mass Storage 设备来说，它们的 `target id` 肯定为 0。有些设备厂家就是要标新立异，它就是要让设备支持多个 `target`，于是它就可以设置 `US_FL_SCM_MULT_TARG` 这么一个 flag，比如我们可以在 `drivers/usb/storage/unusual_devs.h` 中看到如下的定义：

```

416 UNUSUAL_DEV( 0x04e6, 0x0002, 0x0100, 0x0100,
417             "Shuttle",
418             "eUSCSI Bridge",
419             US_SC_DEVICE, US_PR_DEVICE, usb_stor_euscsi_init,
420             US_FL_SCM_MULT_TARG ),

```

然后 352 行，`us->srb->device->lun` 不应该大于 `us->max_lun`，这两个东西是什么区别？`us->max_lun` 是咱们早期在 `usb_stor_scan_thread()` 中调用 `usb_stor_Bulk_max_lun()` 函数来向 usb mass storage 设备获得的最大 LUN，比如 MAX LUN 等于 3，那么这个设备支持的就是 4 个 LUN，

即 0, 1, 2, 3。而 us->srb->device->lun 则可以是这四个值中的任意一个，看传递进来的命令是要访问谁了。但它显然不可能超过 MAX LUN。

然后就是 358 行了。看到这么一个 flag-US\_FL\_FIX\_INQUIRY, 这又是 us->flags 中众多 flag 中的一个，我们前面已经介绍过这个 flag，一些定义于 drivers/usb/storage/unusual\_devs.h 中的设备有这个 flag。事实上，通常大多数设备的厂商名（Vendor Name）和产品名（Product Name）是通过 INQUIRY 命令来获得的，而这个 flag 表明，这些设备的厂商名和产品名不需要查询，或者根本就不支持查询，它们的厂商名和产品名直接就定义好了，在 unusual\_devs.h 中就设好了。那么 358 行这里这个 cmnd[0]是什么？struct scsi\_cmnd 里边有这么一个成员，

```
65 #define MAX_COMMAND_SIZE 16
66     unsigned char cmnd[MAX_COMMAND_SIZE];
```

这个数组 16 个元素，它包含的就是 SCSI 命令，要看懂这个条件判断，得先看下边那句 fill\_inquiry\_response()函数调用。

最后“贴”几个设了 US\_FL\_FIX\_INQUIRY 这个 flag 的设备，这几个都是 Sony 的 PEG 记忆棒，或者叫记忆卡，可以用在 PDA 里边。drivers/usb/storage/unusual\_devs.h 中：

```
635 /* Submitted by Nathan Babb <nathan@lexi.com> */
636 UNUSUAL_DEV( 0x054c, 0x006d, 0x0000, 0x9999,
637             "Sony",
638             "PEG Mass Storage",
639             US_SC_DEVICE, US_PR_DEVICE, NULL,
640             US_FL_FIX_INQUIRY ),
641
642 /* Submitted by Mike Alborn <malborn@deandra.homeip.net> */
643 UNUSUAL_DEV( 0x054c, 0x016a, 0x0000, 0x9999,
644             "Sony",
645             "PEG Mass Storage",
646             US_SC_DEVICE, US_PR_DEVICE, NULL,
647             US_FL_FIX_INQUIRY ),
648
649 /* Submitted by Frank Engel <frankie@cse.unsw.edu.au> */
650 UNUSUAL_DEV( 0x054c, 0x0099, 0x0000, 0x9999,
651             "Sony",
652             "PEG Mass Storage",
653             US_SC_DEVICE, US_PR_DEVICE, NULL,
654             US_FL_FIX_INQUIRY ),
655
```

## 29. 彼岸花的传说（六）

我们继续接着上一节往下看。fill\_inquiry\_response(), 这个函数来自 drivers/usb/storage/usb.c 中。

```

266 void fill_inquiry_response(struct us_data *us, unsigned char *data,
267                          unsigned int data_len)
268 {
269     if (data_len<36) // You lose.
270         return;
271
272     if(data[0]&0x20) { /* USB device currently not connected. Return
273                      peripheral qualifier 001b ("...however, the
274                      physical device is not currently connected
275                      to this logical unit") and leave vendor and
276                      product identification empty. ("If the target
277                      does store some of the INQUIRY data on the
278                      device, it may return zeros or ASCII spaces
279                      (20h) in those fields until the data is
280                      available from the device."). */
281         memset(data+8,0,28);
282     } else {
283         u16 bcdDevice = le16_to_cpu(us->pusb_dev->descriptor.bcdDevice);
284         memcpy(data+8, us->unusual_dev->vendorName,
285               strlen(us->unusual_dev->vendorName) > 8 ? 8 :
286               strlen(us->unusual_dev->vendorName));
287         memcpy(data+16, us->unusual_dev->productName,
288               strlen(us->unusual_dev->productName) > 16 ? 16 :
289               strlen(us->unusual_dev->productName));
290         data[32] = 0x30 + ((bcdDevice>>12) & 0x0F);
291         data[33] = 0x30 + ((bcdDevice>>8) & 0x0F);
292         data[34] = 0x30 + ((bcdDevice>>4) & 0x0F);
293         data[35] = 0x30 + ((bcdDevice) & 0x0F);
294     }
295
296     usb_stor_set_xfer_buf(data, data_len, us->srb);
297 }

```

故事发生得太突然，会让人产生幻觉。本来我们正儿八经用来处理 SCSI 命令的函数是后面将要讲的 `proto_handler()`，但想不到我们在这里开始接触 SCSI 命令了。理由正是因为像 Sony 这几款 PEG 产品做得不好，连最基本的 SCSI 命令 INQUIRY 都不支持，然后又想在 Linux 中使用，那没办法了，所以就准备一个函数来修复这个问题吧，毫无疑问，这属于硬件上的一个 Bug。

什么是 INQUIRY 命令？前面也提过，INQUIRY 命令是最基本的一个 SCSI 命令。比如主机第一次探测设备时就要用 INQUIRY 命令来了解这是一个什么设备，如果 SCSI 总线上有一个插槽插了一个设备，那么 SCSI 主机就问它，你是 SCSI 磁盘，还是 SCSI 磁带，又或是 SCSI 的 CD ROM 呢？作为设备，它内部一定有一段固件程序，即所谓的 `firmware`。它就在接收到主机的 INQUIRY 命令之后做出回答。

具体应该怎么回答？当然是依据 SCSI 协议中规定的格式了。不仅仅 INQUIRY 命令，对于每一个命令都应该如此。只要对方问：“天王盖地虎”。作为设备就该回答：“宝塔镇河妖。”这其实就好比我们对对联，这都是不成文的规矩，而开发 SCSI 的人把这些写成了规范，它就变成了成文的规矩了。具体来说，设备在受到 INQUIRY 命令查询时，它的相应遵从 SCSI 协议中面规定的标准格式，标准格式规定了，响应数据必须至少包含 36 个字节。所以 252 行，如果

`data_len` 小于 36，那就别往下走了，返回吧。

如果你对 SCSI 协议很陌生，还是没有明白 `INQUIRY` 命令究竟是做什么，那么推荐一个工具给你，你可以试一试，以便有一个直观的印象，其实 `INQUIRY` 命令就是查询，查询设备的一些基本信息。从软件的角度来说，在主机扫描时，或者说枚举时，向每一个设备发送这个命令，并且获得回答，驱动程序从此就会保存这些信息，因为这些信息之后可能都会用到或者说其中的一部分会被用到。这里推荐的工具是 `sg_utils3`，这是一个软件包，Linux 中可以使用的软件包，到处都有，下了之后安装上，然后它包含一个应用程序 `sg_inq`，这其实就是给设备发送 `INQUIRY` 命令用的，用法如下所示：

```
[root@localhost ~]# sg_inq -36 /dev/sda
standard INQUIRY:
PQual=0 Device_type=0 RMB=1 version=0x02 [SCSI-2]
[AERC=0] [TrmTsk=0] NormACA=0 HiSUP=0 Resp_data_format=2
SCCS=0 ACC=0 TGPS=0 3PC=0 Protect=0 BQue=0
EncServ=0 MultiP=0 [MChngr=0] [ACKREQQ=0] Addr16=0
[RelAdr=0] WBus16=0 Sync=0 Linked=0 [TranDis=0] CmdQue=0
length=36 (0x24) Peripheral device type: disk
Vendor identification: Intel
Product identification: Flash Disk
Product revision level: 2.00
```

这里我使用的是 Intel 生产的一块 U 盘，使用 `sg_inq` 命令可以查询到关于这块 U 盘的基本信息。实际上 `sg_inq` 可以查询所有 SCSI 设备的信息，因为 `INQUIRY` 本来就是一个标准的 SCSI 命令。当然以上这些信息中，我们之后用得到的大概也就是 Vendor ID，Product ID，Product revision，以及 length，device type--disk，还有中括号里的 SCSI-2，这代表遵守的 SCSI 的版本。SCSI 协议也发展了这么多年，当然也有不同的版本了。

有了直观的印象了我们就继续看代码，272 行，判断 `data[0]` 是否是 20h，20h 有什么特别的吗？当然。SCSI 协议中规定了，标准的 `INQUIRY data` 的 `data[0]`，总共有 8 个 bit。其中 bit7~bit5 被称为 peripheral qualifier（三位），而 bit4~bit0 被称为 peripheral device type（五位），它们代表了不同的含义，但是 20h 就表示 peripheral qualifier 这个外围设备限定符为 001b，而 peripheral device type 这个外围设备类型则为 00h。查阅 SCSI 协议可知，后者代表的是设备类型为磁盘，或者说直接访问设备，前者代表的是目标设备的当前 LUN 支持这种类型。然而，实际的物理设备并没有连接在当前 LUN 上。在 `data[36]` 中，从 `data[8]` 一直到 `data[35]` 这 28 个字节保存的都是厂商和产品的信息。SCSI 协议中写了，如果设备中保存这些信息，那么它可以暂时先返回 0x20h，因为现在是系统 power on 时期或者是 reset 期间，要尽量减少延时，于是 `fill_inquiry_response()` 就会把 `data[8]` 到 `data[35]` 都给设置成 0。等到保存在设备上的这些信息可以读了再去读。

如果不是 20h，比如我们这里传递进来的 `data[0]` 就是 0，那么看 284 行，`data[8]` 开始的 8 个字节可以保存厂商相关的信息，对于，`us->unusual_dev`，我们早已不陌生，`struct us_data` 结构体中的成员 `struct us_unusual_dev *unusual_dev`，我们在 `storage_probe()` 时曾经把

`us_unusual_dev_list[]` 数组中的对应元素赋给了它,而 `us_unusua_dev_list[]` 又来自 `unusual_devs.h`, 都是预先定义好了的。所以这里就是把其中的 `vendorName` 复制到 `data` 数组中来,但是如果 `vendorName` 超过 8 个字符了那可不行,只取前 8 个就行了。当然像 Intel 就不存在这个问题了,只有 5 个字符,大多数公司也都是 8 个字符以内,比如长一点的名字有 Motorola, Samsung 也都没问题。同样 `productName` 也是一样的方法,复制到 `data` 数组中来,协议中规定了,从 16 开始存放 `productName`,不能超过 16 个字符,那么“Flash Disk”也没有问题。关于标准的 INQUIRY 数据格式,可以参看 SCSI Primary Command-4 文档。

然后可以看 290 行, `us->pusb_dev->descriptor.bcdDevice`, `struct us_data` 中有一个成员 `struct usb_device *pusb_dev`, 而 `struct usb_device` 中有一个成员 `struct usb_device_descriptor descriptor`, 而 `struct usb_device_descriptor` 中的成员 `__u16 bcdDevice`, 表示制造商指定的产品的版本号,“道上”的规定是用版本号,制造商 id 和产品 id 来标志一个设备。`bcdDevice` 一共 16 位,是以 bcd 码的方式保存的信息,也就是说,每 4 位代表一个十进制的数,比如 0011 0110 1001 0111 就代表的 3697。而在 SCSI 标准的 INQUIRY data 中, `data[32]` 到 `data[35]` 被定义为保存这四个数,并且要求以 ASCII 码的方式保存。ASCII 码中 48 对应咱们日常的 0, 49 对应 1, 50 对应 2, 也就是说得在现有数字的基础上加上 48, 或者说加上 0x30。这就是 290 行到 293 行所表达的意思。

一切准备好了之后,我们就可以把 `data` 数组,这个包含 36 个字符的信息发送到 SCSI 命令指定的位置了,即 `srb` 指定的位置。这正是 296 行中 `usb_stor_set_xfer_buf` 的所作所为。

在接着讲 296 行这个函数 `usb_stor_set_xfer_buf` 之前,先解释一下之前定义 `data_ptr[36]` 时初始化的前 8 个元素。它们的含义都和 scsi 协议规定的对应。`data_ptr[0]` 不用说了, `data_ptr[1]` 被赋为 0x80, 这表明这个设备是可移除的, `data_ptr[2]` 被赋为 0x02 这说明设备遵循 SCSI-2 协议, `data_ptr[3]` 被赋为 0x02, 说明数据格式遵循国际标准化组织所规定的格式,而 `data_ptr[4]` 被称为 `additional length`, 附加参数的长度,即除了用这么一个标准格式的数据响应之外,可能还会返回更多的一些信息。这里设置的是 0x1F。

---

## 30. 彼岸花的传说 (七)

很显然,我们是把为 INQUIRY 命令准备的数据保存到了我们自己定义的一个结构体中,即 `struct data_ptr[36]`, 但是我们是为了回应一个 SCSI 命令,最终需要知道答案的是 SCSI 核心层。正是它们传递了一个 `scsi_cmnd` 结构体下来,即 `srb`。`struct scsi_cmnd` 中有两个成员,即 `unsigned request_bufflen` 和 `void *request_buffer`, 应该把 `data` 数组中的数据传送到 `request_buffer` 中去,这样,SCSI 核心层就知道去哪里获取结果。没错,当时就是这样!

`usb_stor_set_xfer_buf()` 这个函数来自, `drivers/usb/storage/protocol.c` 中:

```

243 /* Store the contents of buffer into srb's transfer buffer and set the
244  * SCSI residue. */
245 void usb_stor_set_xfer_buf(unsigned char *buffer,
246     unsigned int buflen, struct scsi_cmnd *srb)
247 {
248     unsigned int index = 0, offset = 0;
249
250     usb_stor_access_xfer_buf(buffer, buflen, srb, &index, &offset,
251         TO_XFER_BUF);
252     if (buflen < srb->request_bufflen)
253         srb->resid = srb->request_bufflen - buflen;
254 }

```

主要调用的又是 `usb_stor_access_xfer_buf()` 函数，这个函数也来自同一个文件：  
`drivers/usb/storage/protocol.c`：

```

159 unsigned int usb_stor_access_xfer_buf(unsigned char *buffer,
160     unsigned int buflen, struct scsi_cmnd *srb, unsigned int *index,
161     unsigned int *offset, enum xfer_buf_dir dir)
162 {
163     unsigned int cnt;
164
165     /* If not using scatter-gather, just transfer the data directly.
166      * Make certain it will fit in the available buffer space. */
167     if (srb->use_sg == 0) {
168         if (*offset >= srb->request_bufflen)
169             return 0;
170         cnt = min(buflen, srb->request_bufflen - *offset);
171         if (dir == TO_XFER_BUF)
172             memcpy((unsigned char *) srb->request_buffer + *offset,
173                 buffer, cnt);
174         else
175             memcpy(buffer, (unsigned char *) srb->request_buffer +
176                 *offset, cnt);
177         *offset += cnt;
178
179     /* Using scatter-gather. We have to go through the list one entry
180      * at a time. Each s-g entry contains some number of pages, and
181      * each page has to be kmap()'ed separately. If the page is already
182      * in kernel-addressable memory then kmap() will return its address.
183      * If the page is not directly accessible -- such as a user buffer
184      * located in high memory -- then kmap() will map it to a temporary
185      * position in the kernel's virtual address space. */
186     } else {
187         struct scatterlist *sg =
188             (struct scatterlist *) srb->request_buffer
189             + *index;
190
191         /* This loop handles a single s-g list entry, which may
192          * include multiple pages. Find the initial page structure
193          * and the starting offset within the page, and update
194          * the *offset and *index values for the next loop. */
195         cnt = 0;
196         while (cnt < buflen && *index < srb->use_sg) {
197             struct page *page = sg->page +
198                 ((sg->offset + *offset) >> PAGE_SHIFT);
199             unsigned int poff =
200                 (sg->offset + *offset) & (PAGE_SIZE-1);

```



```

201         unsigned int sglen = sg->length - *offset;
202
203         if (sglen > buflen - cnt) {
204
205             /* Transfer ends within this s-g entry */
206             sglen = buflen - cnt;
207             *offset += sglen;
208         } else {
209
210             /* Transfer continues to next s-g entry */
211             *offset = 0;
212             ++*index;
213             ++sg;
214         }
215
216         /* Transfer the data for all the pages in this
217          * s-g entry. For each page: call kmap(), do the
218          * transfer, and call kunmap() immediately after. */
219         while (sglen > 0) {
220             unsigned int plen = min(sglen, (unsigned int)
221                                     PAGE_SIZE - poff);
222             unsigned char *ptr = kmap(page);
223
224             if (dir == TO_XFER_BUF)
225                 memcpy(ptr + poff, buffer + cnt, plen);
226             else
227                 memcpy(buffer + cnt, ptr + poff, plen);
228             kunmap(page);
229
230             /* Start at the beginning of the next page */
231             poff = 0;
232             ++page;
233             cnt += plen;
234             sglen -= plen;
235         }
236     }
237 }
238
239 /* Return the amount actually transferred */
240 return cnt;
241 }

```

在编写 Linux 设备驱动时，总是要涉及内存管理。内存管理毫无疑问是 Linux 内核中最复杂的一部分，能不涉及我们都希望别去涉及。但生活中总是充满了无奈，该来的还是会来。

所以，usb\_stor\_access\_xfer\_buf()函数映入了我们的眼帘。

首先判断 `srb->use_sg` 是否为 0。IT 玩家们创建了一个词，即 `scatter/gather`，它是一种用于高性能 IO 的标准技术。它通常意味着一种 DMA 传输方式，对于一个给定的数据块，它可能在内存中存在一些离散的缓冲区。换言之，就是一些不连续的内存缓冲区一起保存一个数据块。如果没有 `scatter/gather`，那么当我们要建立一个从内存到磁盘的传输，那么操作系统通常会为每一个 buffer 做一次传输，或者干脆就是把这些不连续的 buffer 里边的东西全都移动到另一个很大的 buffer 里面，再开始传输。那么这两种方法显然都是效率不高的。

毫无疑问，如果操作系统、驱动程序或硬件能够把这些来自内存中离散位置的数据收集起来（gather up）并转移它们到适当位置整个这个步骤是一个单一的操作的话，效率肯定就会更高。反之，如果要从磁盘向内存中传输，而有一个单一的操作能够把数据块直接分散开来（scatter）到达内存中需要的位置，而不再需要中间的那个块移动，或者别的方法，那么显然，效率总会更高。

在 struct scsi\_cmnd 中，有一个成员 unsigned short use\_sg，上头传下来的 scsi\_cmnd，其 use\_sg 是设好了的，这里判断一下。如果它为 0，那么说明没有使用 scatter/gather。struct scsi\_cmnd 中还有两个成员，unsigned request\_bufflen 和 void \*request\_buffer，它们和 use\_sg 是什么关系呢？

事实上，要使用 scatter/gather，就需要一个 scatterlist 数组，有人称它为散列表数组。对于不同的硬件平台，定义了不同的 struct scatterlist 结构体，它们来自 include/asm/scatterlist.h 中。（如果是硬件平台 i386 的，那么就是 include/asm-i386/scatterlist.h，如果是 x86\_64 的平台，那么就在 include/asm-x86\_64/scatterlist.h 中），然后所谓的 scatter/gather 就是一次把整个 scatterlist 数组给传送掉。而 use\_sg 为 0 就表示没有 scatter gather list，或者说 scatterlist，对于这种情况，数据将直接传送给 request\_buffer 或者直接从 request\_buffer 中取得数据。而如果 use\_sg 大于 0，那么表示 scatter gather list 这么一个数组就在 request\_buffer 中，而数组元素个数正是 use\_sg 个。也就是说，srb->request\_buffer 里边的数据有两种可能，一种是包含了数据本身，另一种是包含了 scatter gather list。具体是哪种情况通过判断 use\_sg 来决定。而接下来即将要讲到的 srb->request\_bufflen 顾名思义，就是 buffer 的长度，但对于 use\_sg 大于 0 的情况，换言之，对于使用 scatter gather list 的情况，request\_bufflen 没有意义，将被忽略。

对这些原理有了基本的了解之后，我们可以从下节开始看代码了。这里先提醒一下，要注意我们这个函数虽然看似是传输数据，可它实际上并没有和 USB 真正发生关系，我们只是从软件上来 fix 一个硬件的 bug，这个 bug 就是我们已经说过了的，不能响应基本的 SCSI 命令 INQUIRY。

所以对于那些不能响应 INQUIRY 命令的设备，当上层的驱动程序去 INQUIRY 时，实际上是调用我们的 queuecommand，那么我们根本就不用和下面的硬件去打交道，就直接回复上层，即我们从软件上来准备这个一段 INQUIRY 数据给上层，这才是我们这个函数的目的。真正的和硬件打交道的代码在后面，我们还没走到那一步。

## 31. 彼岸花的传说（八）

对于 use\_sg 为 0 的情况，我们接下来再看 168 行，offset 是函数调用传递进来的参数，注释里说得很清楚，就是用来标志偏移量的，每次复制几个字节它就增加几个字节，最大它也不

能超过 `request_bufflen`，这是显然的。`usb_stor_access_xfer_buf()`这个函数所做的事情就是从 `srb->request_buffer` 往 `buffer` 里边复制数据，或者反过来从 `buffer` 往 `srb->request_buffer`，然后返回复制了多少个字节。对于 `offset` 大于等于 `request_bufflen` 的情况，当然就直接返回 0 了，因为 `request_buffer` 已经满了。

参数 `enum xfer_buf_dir dir` 标志的正是传输方向，这个数据类型是在 `drivers/usb/storage/protocol.h` 中被定义的：

```
51 /* struct scsi_cmnd transfer buffer access utilities */
52 enum xfer_buf_dir {TO_XFER_BUF, FROM_XFER_BUF};
```

这个参数其实是很简单的一个枚举数据类型，含义也很简单：一个表示向 `srb->request_buffer` 里边复制，`TO_XFER_BUF`；另一个表示从 `srb->request_buffer` 里边往外复制，`FROM_XFER_BUF`。（题外话：XFER 就是 TRANSFER 的意思，外国人喜欢这样缩写。刚进 Intel 时老板专门给了我一个 Excel 文件，里边全是 Intel 内部广泛使用的英文缩写，不在 Intel 待一段时间基本没法理解。）其中定义成枚举数据类型也是很有必要的，因为数据传输肯定得有且仅有一个方向。而此情此景，我们传进来的是前者，所以 171 行判断之后会执行 172 行，从 `buffer` 里边复制 `cnt` 个字节到 `(unsigned char *)srb->request_buffer + *offset` 去。`cnt` 在 170 行确定，`min` 函数不用说也知道，取最小值，不过 linux 内核中确实有定义这个函数，在 `include/linux/kernel.h` 中。

而 170 行比较的是一个 `buflen`，一个 `srb->request_bufflen-*offset`，咱们这次要传送的数据长度是 `buflen`，但是显然也不能够超过后者，所以就取其中小的值，调用 `memcpy` 复制，然后 177 行 `*offset` 加上复制的字节 `cnt`，对于这种不采用 `scatter gather` 方式的传输，那么到这里就可以返回了，直接就到 240 行，返回 `cnt` 即可。

但是对于使用 `scatter gather` 方式的传输，情况当然不一样了。从 186 行开始往下看，显然如我们所说，得定义一个 `struct scatterlist` 结构体的指针，由于 `struct scatterlist` 是和体系结构有关的，以 i386 的为例，`include/asm-i386/scatterlist.h`：

```
6 struct scatterlist {
7     struct page      *page;
8     unsigned int      offset;
9     dma_addr_t        dma_address;
10    unsigned int      length;
11 };
```

这个结构并不复杂，其中 `page` 指针通常指向将要进行 `scatter gather` 操作的 `buffer`。而 `length` 表示 `buffer` 的长度，`offset` 表示 `buffer` 在 `page` 内的偏移量。187 行，定义一个 `struct scatterlist` 指针 `sg`，然后令它指向 `(struct scatterlist*)srb->request_buffer+*index`，搜索一下内核代码就可以知道，每次 `*index` 都是被初始化为 0 然后才调用 `usb_stor_access_xfer_buf()` 的。195 行，`cnt` 设为 0，196 行开始进入循环，循环的条件是 `cnt` 小于 `buflen`，同时 `*index` 小于 `srb->use_sg`，`srb->use_sg` 在前面说过了，只要它不是 0，那么它里边的值就代表了 `scatter gather` 传输时数组元素的个数。

请注意，187 行这里让 `sg` 等于 `srb->request_buffer`，（当然还要加上 `*index`，如果 `index` 不为 0 的话），那么 `request_buffer` 究竟是什么？对于使用 `scatter/gather` 传输的情况，`request_buffer` 里边实际上是一个数组，每一个元素都是一个 `struct scatterlist` 的指针，而每一个 `scatterlist` 指针的 `page` 里边包含了一些 `page`（而不是一个 `page`），而 `offset` 里边包含的是每一个 DMA buffer 的总的偏移量，它由两部分组成，高位部分标志着 `page` 号，低位部分标志着具体某个 `page` 中的偏移量，高位低位由 `PAGE_SHIFT` 宏来划分。不同的硬件平台 `PAGE_SHIFT` 值不一样，因为不同的硬件平台 `page` 的大小也不一样，即这里的 `PAGE_SIZE` 不一样。目前比较前卫的硬件平台其 `page size` 有 4KB 或者 8KB 的，而 `PAGE_SHIFT` 也就是 12 或者 13。换言之，`sg->offset` 去掉低 12 位或者低 13 位就是 `page` 号，而低 12 位或者低 13 位恰恰是在该 `page` 内的偏移量。之所以可以把一个 `sg->offset` 起两个作用，正是因为 `page size` 只需要 12 位或者 13 位就足够了，或者说偏移量本身只有 12 位或者 13 位，而一个 `int` 型变量显然可以包含比 12 位或 13 位更多的信息。我们最终是要把 `buffer`（即前面说的那个 36 个 Bytes 的标准的 INQUIRY data buffer）里边的信息复制至 DMA buffer 中，`buffer` 我们已经知道，它就是 `usb_stor_access_xfer_buf()` 函数传递进来的参数，而 DMA buffer 在哪呢？只要我们知道它在哪个 `page` 中，知道它的 `offset`，那么有一个函数可以帮助我们获得它对应的内核虚拟地址，而这个地址正是我们需要的，有了它，我们就能调用 `memcpy` 函数来复制数据了。

所以 197 行到 200 行，就是计算出究竟是哪个 `page`，究竟是多少 `offset`，后者用被赋给了 `unsigned int` 变量 `poff`，`*offset` 是 `usb_stor_access_xfer_buf()` 函数传递进来的参数，它也可以控制我们要传送数据的 DMA buffer 对应的 `offset`，不过这里我们传递进来的是 0。所以不去关注。

201 行对 `unsigned int sglen` 赋值，`sg->length` 实际上就是 DMA buffer 的长度。所以显然我们复制的数据不能超过这个长度。如果我们还指定了 `*offset`，就表明 DMA buffer 中从 `*offset` 开始装，那么就不能超过 `sg->length-*offset`。

203 行，由于我们现在还在 `while` 循环中，所以先看第一次执行到 203 行，这时 `cnt` 等于 0，`buflen` 就是 data buffer 的长度，传进来的是 36。`sglen` 表示了 DMA buffer 里面可以装多少数据，如果 `sglen` 比 `buflen` 要大，那么很好，一次就可以装满。因为这就好比 `buflen` 是一吨沙子，而 `sglen` 则表示装沙车载重两吨或者更多，比如三吨。这样 206 行和 207 行的作用就是做一个标记，比如 `sglen` 被用来记录实际装载了多少重量的沙子，而 `*offset` 则表示了装沙车用了多少了，如果还没卸货下次又要继续往里装那就装吧，反正没满就可以装。如果 `sglen` 比 `buflen` 要小或者刚好够大，那么 `*offset` 肯定就被设为 0，因为这一车必然会被装满，要再装沙子只能调用下一辆车，所以同时 `sg` 和 `*index` 也自加。

然后来看 219 行了，`sglen` 一开始肯定应该大于 0，它表示的是实际装了多少数据，但是内存管理机制有一个限制，`memcpy` 传输不能够跨页，也就是说不能跨 `page`，一次最多就是把本 `page` 的内容复制，如果你的数据超过了一个 `page`，那你还得再复制一次或者多次，因为这里使用了循环。

每一次真正复制的长度用 `plen` 来表示，所以 233 行和 234 行，`cnt` 是计数的，所以要加上一个 `plen`，而 `sglen` 也是一个计数的，但是它是反着计，所以它每次要减掉一个 `plen`。而 `poff` 和 `page` 一个设为 0，一个自加，这个道理很简单，从下一页开头进行继续复制。而 220 行再次调用强大的 `min()` 函数也正是为了保证每次复制不能跨页。

222 行和 228 行 `kmap()` 和 `kunmap()` 出现了，道理也很简单。这对冤家是内核中的内存管理部门提供的重要函数，其中 `kmap()` 函数的作用就是传递给它一个 `struct page` 指针，它能返回这个 `page` 指针所指的 `page` 的内核虚拟地址，而有了这个 `page` 对应的虚拟地址，加上前面已经知道的偏移量，就可以调用 `memcpy` 函数来复制数据了。至于 `kunmap`，凡是 `kmap()` 建立起来的良好关系必须由 `kunmap()` 来摧毁。

然后，224 行到 227 行的两个 `memcpy` 无须再讲了。

240 行，返回实际传输的字节数。

至此，`usb_stor_access_xfer_buf()` 这个函数都讲完了。（注：应该说如果不是很有悟性的话，这段代码要看懂还是挺难的，要理解这个函数，必须明白这个双重循环，须知外循环是按 `sg entry` 来循环的，即一个 `sg entry` 循环一次，而内循环是按 `page` 来循环的，我们说过，一个 `sg entry` 可以有多个 `page`，因为没有办法跨 `page` 映射，所以只能每个 `page` 映射一次，所以才需要循环。）

回到 `usb_stor_set_xfer_buf()` 中来，也只剩下一句话了，如果我们要传递的 36 个字节还不足以填满这辆装沙车的话，那就让我们记录下这辆车还能装多少沙子吧，`srb->resid` 正是扮演着这个记录者的角色。

然后我们回到 `fill_inquiry_response()` 中来，这个函数也结束了。我们再一次回到 `usb_stor_control_thread()` 中来，这个函数总是隔一会儿又会出现。对于 `INQUIRY` 命令，咱们在 `fill_inquiry_response()` 之后，把 `srb->result` 设为了 `SAM_STAT_GOOD`，它是 `scsi` 系统里面定义的一个宏，`include/scsi/scsi.h` 中：

```
129 #define SAM_STAT_GOOD          0x00
```

其实就是 0，完成了工作之后把 `srb->result` 设为 0，以后 `scsi` 那边的函数会去检测，不用咱们管了。

然后我们进入到 384 行，更确切地说是 388 行，`srb->scsi_done()` 函数会被调用。`srb->scsi_done()` 实际上是一个函数指针，在 `queuecommand` 中令它等于 `scsi_done`，我们被要求在完成了该做的事情之后调用这个函数，剩下的事情 `SCSI` 核心层会去处理。

针对 `queuecommand` 传递进来 `INQUIRY` 命令的情况，该做的就都做了。

最后单独解释一下：

首先,好端端的传输数据为什么要分散成好多个 scatter gather list,这不是自找麻烦吗? SCSI 层包括 usb-storage 之所以要使用 scatter gather,是因为这个特性允许系统在一个 SCSI 命令中传输多个物理上不连续的 buffers。

kmap()和 kunmap()这两个函数是干什么的?为什么要映射?这个世界上有一个地址,叫做物理地址,还有一个地址叫做内核地址。struct page 代表的是物理地址,内核用这个结构体来代表每一个物理 page,或者说物理页,显然我们代码中不能直接操作物理地址,memcpy 这个函数根本就不认识物理地址,它只能使用内核地址。所以我们需要把物理地址映射到内核地址空间中来。kmap()从事的正是这项伟大的工作。不过写过代码的人了解 kmap()更多地是在和 high memory 打交道时认识的。

## 32. 彼岸花的传说 (The End)

解决了这个 INQUIRY 的问题,我们就可以继续往下走了,372 行,这就是真正的批量传输的地方,proto\_handler()就是正儿八经的处理 SCSI 命令的函数指针。而 usb\_stor\_control\_thread 之前的所有代码就是为了判断是不是有必要调用函数 proto\_handler(),比如超时了,比如模块该卸载了,比如设置断开 flag 了,比如要处理的就是这个有问题的 INQUIRY 等,这些情况都需要先排除了才有必要到达这里来执行真正的命令。实际上这就是先从宏观上来控制,保证我们走的是一条正确的道路,而不至于沿着错误的道路走半天。毕竟,在错误的路上,就算奔跑也没有用!

我们倒是先不急着想到 proto\_handler 里边去看,先把外边的代码看完。小时候,我们不都天真地以为,外面的世界很精彩吗?我们先跳过 proto\_handler(),把 usb\_stor\_control\_thread()中剩下的代码看完,从而完整地了解这个守护进程究竟是如何循环的。

385 行,只要刚才的命令的结果即 srb->result 不为 DID\_ABORT,那么就是成功执行了。于是我们就调用 scsi\_done 函数。

390 行,SkipForAbort,就是一个行标志,对应前面的 goto SkipForAbort 语句。继续说,399 行,前面的注释也说得清楚了,如果是设置了 US\_FLIDX\_TIMED\_OUT 那么就唤醒设这个 flag 的进程,其实就是唤醒 command\_abort,后面我们会讲 command\_abort()。这里之所以判断这个 flag 而不是判断 srb->result==DID\_ABORT 注释里说得也很清楚,因为有可能是在 usb 传输结束之后才收到的 abort 命令,换言之,即便你的 srb->result 不为 DID\_ABORT 也可能最新又接到了 abort 的请求,所以这里就判断 abort 请求必然要设置的一个 flag 来判断。

408 行,如果要断开了,或者是一个命令执行完了,或者是 abort 了,那么最终就是把 us->srb

置空。剩下两行的两把锁我们已经说过，会到最后统一讲。

413 行，至此，这个守护进程就算是走了一遍，for 循环继续。

最后，只剩下 431 这行了，程序执行到这一行意味着 for 循环结束了，而从 for 循环的代码我们不难看出，结束 for 循环的只有一句话，就是 break。前面说过，它意味着模块要被卸载了。所以这里 complete\_and\_exit() 就是唤醒别人同时结束自己，于是这一刻，usb\_stor\_control\_thread() 也就正式结束了，也许对它来说，结束就是解脱吧。

关于这个守护进程，我们也终于讲完了，其中的函数 proto\_handler() 这两行，因为其重要性，我们单独挑出来讲。

## 33. SCSI 命令之我型我秀

usb\_stor\_control\_thread() 基本讲完了，但是其中下面这几行，正是高潮中的高潮。所谓的批量传输，所谓的 Bulk-Only 协议。正是在这里体现出来的。

```
371         /* we've got a command, let's do it! */
372         else {
373             US_DEBUG(usb_stor_show_command(us->srb));
374             us->proto_handler(us->srb, us);
375         }
```

所谓的 US\_DEBUG，我们前面已经讲过，无非就是打印条是信息的。而眼下这句话就是执行 usb\_stor\_show\_command(us->srb) 这个函数，鉴于这个函数是我们自己写的，而且有意义，所以也就列出来。这个函数定义于 drivers/usb/storage/debug.c 中：

```
56 void usb_stor_show_command(struct scsi_cmnd *srb)
57 {
58     char *what = NULL;
59     int i;
60
61     switch (srb->cmnd[0]) {
62     case TEST_UNIT_READY: what = "TEST_UNIT_READY"; break;
63     case REZERO_UNIT: what = "REZERO_UNIT"; break;
64     case REQUEST_SENSE: what = "REQUEST_SENSE"; break;
65     case FORMAT_UNIT: what = "FORMAT_UNIT"; break;
66     case READ_BLOCK_LIMITS: what = "READ_BLOCK_LIMITS"; break;
67     case REASSIGN_BLOCKS: what = "REASSIGN_BLOCKS"; break;
68     case READ_6: what = "READ_6"; break;
69     case WRITE_6: what = "WRITE_6"; break;
70     case SEEK_6: what = "SEEK_6"; break;
71     case READ_REVERSE: what = "READ_REVERSE"; break;
72     case WRITE_FILEMARKS: what = "WRITE_FILEMARKS"; break;
73     case SPACE: what = "SPACE"; break;
74     case INQUIRY: what = "INQUIRY"; break;
```

```

75     case RECOVER_BUFFERED_DATA: what = "RECOVER_BUFFERED_DATA"; break;
76     case MODE_SELECT: what = "MODE_SELECT"; break;
77     case RESERVE: what = "RESERVE"; break;
78     case RELEASE: what = "RELEASE"; break;
79     case COPY: what = "COPY"; break;
80     case ERASE: what = "ERASE"; break;
81     case MODE_SENSE: what = "MODE_SENSE"; break;
82     case START_STOP: what = "START_STOP"; break;
83     case RECEIVE_DIAGNOSTIC: what = "RECEIVE_DIAGNOSTIC"; break;
84     case SEND_DIAGNOSTIC: what = "SEND_DIAGNOSTIC"; break;
85     case ALLOW_MEDIUM_REMOVAL: what = "ALLOW_MEDIUM_REMOVAL"; break;
86     case SET_WINDOW: what = "SET_WINDOW"; break;
87     case READ_CAPACITY: what = "READ_CAPACITY"; break;
88     case READ_10: what = "READ_10"; break;
89     case WRITE_10: what = "WRITE_10"; break;
90     case SEEK_10: what = "SEEK_10"; break;
91     case WRITE_VERIFY: what = "WRITE_VERIFY"; break;
92     case VERIFY: what = "VERIFY"; break;
93     case SEARCH_HIGH: what = "SEARCH_HIGH"; break;
94     case SEARCH_EQUAL: what = "SEARCH_EQUAL"; break;
95     case SEARCH_LOW: what = "SEARCH_LOW"; break;
96     case SET_LIMITS: what = "SET_LIMITS"; break;
97     case READ_POSITION: what = "READ_POSITION"; break;
98     case SYNCHRONIZE_CACHE: what = "SYNCHRONIZE_CACHE"; break;
99     case LOCK_UNLOCK_CACHE: what = "LOCK_UNLOCK_CACHE"; break;
100    case READ_DEFECT_DATA: what = "READ_DEFECT_DATA"; break;
101    case MEDIUM_SCAN: what = "MEDIUM_SCAN"; break;
102    case COMPARE: what = "COMPARE"; break;
103    case COPY_VERIFY: what = "COPY_VERIFY"; break;
104    case WRITE_BUFFER: what = "WRITE_BUFFER"; break;
105    case READ_BUFFER: what = "READ_BUFFER"; break;
106    case UPDATE_BLOCK: what = "UPDATE_BLOCK"; break;
107    case READ_LONG: what = "READ_LONG"; break;
108    case WRITE_LONG: what = "WRITE_LONG"; break;
109    case CHANGE_DEFINITION: what = "CHANGE_DEFINITION"; break;
110    case WRITE_SAME: what = "WRITE_SAME"; break;
111    case GPCMD_READ_SUBCHANNEL: what = "READ SUBCHANNEL"; break;
112    case READ_TOC: what = "READ_TOC"; break;
113    case GPCMD_READ_HEADER: what = "READ HEADER"; break;
114    case GPCMD_PLAY_AUDIO_10: what = "PLAY AUDIO (10)"; break;
115    case GPCMD_PLAY_AUDIO_MSF: what = "PLAY AUDIO MSF"; break;
116    case GPCMD_GET_EVENT_STATUS_NOTIFICATION:
117        what = "GET EVENT/STATUS NOTIFICATION"; break;
118    case GPCMD_PAUSE_RESUME: what = "PAUSE/RESUME"; break;
119    case LOG_SELECT: what = "LOG_SELECT"; break;
120    case LOG_SENSE: what = "LOG_SENSE"; break;
121    case GPCMD_STOP_PLAY_SCAN: what = "STOP PLAY/SCAN"; break;
122    case GPCMD_READ_DISC_INFO: what = "READ DISC INFORMATION"; break;
123    case GPCMD_READ_TRACK_RZONE_INFO:
124        what = "READ TRACK INFORMATION"; break;
125    case GPCMD_RESERVE_RZONE_TRACK: what = "RESERVE TRACK"; break;
126    case GPCMD_SEND_OPC: what = "SEND OPC"; break;
127    case MODE_SELECT_10: what = "MODE_SELECT_10"; break;
128    case GPCMD_REPAIR_RZONE_TRACK: what = "REPAIR TRACK"; break;
129    case 0x59: what = "READ MASTER CUE"; break;
130    case MODE_SENSE_10: what = "MODE_SENSE_10"; break;
131    case GPCMD_CLOSE_TRACK: what = "CLOSE TRACK/SESSION"; break;
132    case 0x5C: what = "READ BUFFER CAPACITY"; break;
133    case 0x5D: what = "SEND CUE SHEET"; break;

```



```
134     case GPCMD_BLANK: what = "BLANK"; break;
135     case REPORT_LUNS: what = "REPORT LUNS"; break;
136     case MOVE_MEDIUM: what = "MOVE_MEDIUM or PLAY AUDIO (12)"; break;
137     case READ_12: what = "READ_12"; break;
138     case WRITE_12: what = "WRITE_12"; break;
139     case WRITE_VERIFY_12: what = "WRITE_VERIFY_12"; break;
140     case SEARCH_HIGH_12: what = "SEARCH_HIGH_12"; break;
141     case SEARCH_EQUAL_12: what = "SEARCH_EQUAL_12"; break;
142     case SEARCH_LOW_12: what = "SEARCH_LOW_12"; break;
143     case SEND_VOLUME_TAG: what = "SEND_VOLUME_TAG"; break;
144     case READ_ELEMENT_STATUS: what = "READ_ELEMENT_STATUS"; break;
145     case GPCMD_READ_CD_MSF: what = "READ CD MSF"; break;
146     case GPCMD_SCAN: what = "SCAN"; break;
147     case GPCMD_SET_SPEED: what = "SET CD SPEED"; break;
148     case GPCMD_MECHANISM_STATUS: what = "MECHANISM STATUS"; break;
149     case GPCMD_READ_CD: what = "READ CD"; break;
150     case 0xE1: what = "WRITE CONTINUE"; break;
151     case WRITE_LONG_2: what = "WRITE_LONG_2"; break;
152     default: what = "(unknown command)"; break;
153 }
154 US_DEBUGP("Command %s (%d Bytes)\n", what, srb->cmd_len);
155 US_DEBUGP("");
156 for (i = 0; i < srb->cmd_len && i < 16; i++)
157     US_DEBUGPX(" %02x", srb->cmnd[i]);
158     US_DEBUGPX("\n");
159 }
```

这个函数，很简单，就是把要执行的 SCSI 命令打印出来。列出这个函数没别的意思，让不熟悉 SCSI 的读者知道基本上会遇到一些什么命令。显然，刚才说的那个 INQUIRY 也包含在其中。

不过别看这个函数很简单，你要是不熟悉 SCSI 协议的话，你还真的解释不了这个函数。比如你说 `srb->cmnd[]` 这个数组到底是什么内容？有什么格式？为什么函数一开始只判断 `cmnd[0]`？实不相瞒，这里边还真有学问。首先，在 SCSI 的规范里边定义了一些命令，每个命令都有一定的格式，命令的字节数也有好几种，有的命令是 6 个字节的，有的命令是 10 个字节的，有的命令是 12 个字节的。如图 4.33.1、图 4.33.2 和图 4.33.3 所示，SCSI 命令就该是这个样子。

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	Reserved			(MSB)				
2	LOGICAL BLOCK ADDRESS (if required)							
3								
4	TRANSFER LENGTH (if required) PARAMETER LIST LENGTH (if required) ALLOCATION LENGTH (if required)							
5	CONTROL							

图 4.33.1 典型的 6 字节命令块描述符

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	Reserved			SERVICE ACTION (if required)				
2	(MSB)							
3								
4	LOGICAL BLOCK ADDRESS (if required)							
5								
6	(LSB)							
7	Reserved							
8	(MSB)							
9	TRANSFER LENGTH (if required)							
	PARAMETER LIST LENGTH (if required)							
	ALLOCATION LENGTH (if required)							
	(LSB)							
9	CONTROL							

图 4.33.2 典型的 10 字节命令块描述符

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	Reserved			miscellaneous CDB information				
2	(MSB)	LOGICAL BLOCK ADDRESS						
3								
4								
5								
6								
7								
8								
9								
10	(MSB)	TRANSFER LENGTH (if required) PARAMETER LIST LENGTH (if required) ALLOCATION LENGTH (if required)						
11								
12								
13		(LSB)						
14	Reserved							
15	CONTROL							

图 4.33.3 典型的 12 字节命令块描述符

人们把这样几个字节的命令称之为 CDB（Command Descriptor Block，命令描述符块）。而我们为 CDB 准备了一个字符数组，结构体 struct scsi\_cmnd 中的 unsigned char cmnd[16]，最大就 12 个字节，为什么不申请一个 12 个字节的数组？

既然这个 CDB 有 16 个字节，那么为什么我们每次都判断 cmnd[0]就够了？仔细看这三幅图，注意到 Operation code 了吗？没错，三幅图中的第 1 个字节都被称为 Operation code，换言之，不管是什么样子的命令，都必须在第 1 个字节里签上自己的名字，向世人说明你是谁。于是在 include/scsi/scsi.h 中，定义了好多宏，比如#define INQUIRY 0x12, #define READ\_6 0x08, #define FORMAT\_UNIT 0x04，实际上操作码就相当于 SCSI 命令的序列号，SCSI 命令总共也就那么多，8 位的操作码已经足够表示了，因此，我们只要用一个字节就可以判断出这是哪个命令了。

好了，命令说完了，开始进入真正处理命令的部分了。

## 34. 迷雾重重的批量传输（一）

374 行，`us->proto_handler()`其实是一个函数指针，知道它指向什么吗？我们早在 `storage_probe()` 中，确切地说，在 `get_protocol()`就赋了值，当时只知道是 `get protocol`，却不知道究竟干什么用，现在该用上了，一个指针要是没什么用人家才不会为它赋值呢。当初我们就讲了，对于 U 盘，`proto_handler` 被赋值为 `usb_stor_transparent_scsi_command`，所以我们来看后者吧。后者定义于 `drivers/usb/storage/protocol.c`：

```
140 void usb_stor_transparent_scsi_command(struct scsi_cmnd *srb,
141                                     struct us_data *us)
142 {
143     /* send the command to the transport layer */
144     usb_stor_invoke_transport(srb, us);
145 }
```

`usb_stor_invoke_transport()`这个函数可不简单。咱们先做好思想准备，接下来就去见识一下它的庐山真面目。它来自 `drivers/usb/storage/transport.c`：

```
505 void usb_stor_invoke_transport(struct scsi_cmnd *srb, struct us_data *us)
506 {
507     int need_auto_sense;
508     int result;
509
510     /* send the command to the transport layer */
511     srb->resid = 0;
512     result = us->transport(srb, us);
513
514     /* if the command gets aborted by the higher layers, we need to
515      * short-circuit all other processing
516      */
517     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
518         US_DEBUGP("-- command was aborted\n");
519         srb->result = DID_ABORT << 16;
520         goto Handle_Errors;
521     }
522
523     /* if there is a transport error, reset and don't auto-sense */
524     if (result == USB_STOR_TRANSPORT_ERROR) {
525         US_DEBUGP("-- transport indicates error, resetting\n");
526         srb->result = DID_ERROR << 16;
527         goto Handle_Errors;
528     }
529
530     /* if the transport provided its own sense data, don't auto-sense */
531     if (result == USB_STOR_TRANSPORT_NO_SENSE) {
532         srb->result = SAM_STAT_CHECK_CONDITION;
533         return;
```

```

534     }
535
536     srb->result = SAM_STAT_GOOD;
537
538     /* Determine if we need to auto-sense
539     *
540     * I normally don't use a flag like this, but it's almost impossible
541     * to understand what's going on here if I don't.
542     */
543     need_auto_sense = 0;
544
545     /*
546     * If we're running the CB transport, which is incapable
547     * of determining status on its own, we will auto-sense
548     * unless the operation involved a data-in transfer. Devices
549     * can signal most data-in errors by stalling the bulk-in pipe.
550     */
551     if ((us->protocol == US_PR_CB || us->protocol == US_PR_DPCM_USB) &&
552         srb->sc_data_direction != DMA_FROM_DEVICE) {
553         US_DEBUGP("-- CB transport device requiring auto-sense\n");
554         need_auto_sense = 1;
555     }
556
557     /*
558     * If we have a failure, we're going to do a REQUEST_SENSE
559     * automatically. Note that we differentiate between a command
560     * "failure" and an "error" in the transport mechanism.
561     */
562     if (result == USB_STOR_TRANSPORT_FAILED) {
563         US_DEBUGP("-- transport indicates command failure\n");
564         need_auto_sense = 1;
565     }
566
567     /*
568     * A short transfer on a command where we don't expect it
569     * is unusual, but it doesn't mean we need to auto-sense.
570     */
571     if ((srb->resid > 0) &&
572         !((srb->cmnd[0] == REQUEST_SENSE) ||
573           (srb->cmnd[0] == INQUIRY) ||
574           (srb->cmnd[0] == MODE_SENSE) ||
575           (srb->cmnd[0] == LOG_SENSE) ||
576           (srb->cmnd[0] == MODE_SENSE_10))) {
577         US_DEBUGP("-- unexpectedly short transfer\n");
578     }
579
580     /* Now, if we need to do the auto-sense, let's do it */
581     if (need_auto_sense) {
582         int temp_result;
583         void* old_request_buffer;
584         unsigned short old_sg;
585         unsigned old_request_bufflen;
586         unsigned char old_sc_data_direction;
587         unsigned char old_cmd_len;
588         unsigned char old_cmnd[MAX_COMMAND_SIZE];
589         int old_resid;
590
591         US_DEBUGP("Issuing auto-REQUEST_SENSE\n");
592     }

```

```

593     /* save the old command */
594     memcpy(old_cmnd, srb->cmnd, MAX_COMMAND_SIZE);
595     old_cmd_len = srb->cmd_len;
596
597     /* set the command and the LUN */
598     memset(srb->cmnd, 0, MAX_COMMAND_SIZE);
599     srb->cmnd[0] = REQUEST_SENSE;
600     srb->cmnd[1] = old_cmnd[1] & 0xE0;
601     srb->cmnd[4] = 18;
602
603     /* FIXME: we must do the protocol translation here */
604     if (us->subclass == US_SC_RBC || us->subclass == US_SC_SCSI)
605         srb->cmd_len = 6;
606     else
607         srb->cmd_len = 12;
608
609     /* set the transfer direction */
610     old_sc_data_direction = srb->sc_data_direction;
611     srb->sc_data_direction = DMA_FROM_DEVICE;
612
613     /* use the new buffer we have */
614     old_request_buffer = srb->request_buffer;
615     srb->request_buffer = us->sensebuf;
616
617     /* set the buffer length for transfer */
618     old_request_bufflen = srb->request_bufflen;
619     srb->request_bufflen = US_SENSE_SIZE;
620
621     /* set up for no scatter-gather use */
622     old_sg = srb->use_sg;
623     srb->use_sg = 0;
624
625     /* issue the auto-sense command */
626     old_resid = srb->resid;
627     srb->resid = 0;
628     temp_result = us->transport(us->srb, us);
629
630     /* let's clean up right away */
631     memcpy(srb->sense_buffer, us->sensebuf, US_SENSE_SIZE);
632     srb->resid = old_resid;
633     srb->request_buffer = old_request_buffer;
634     srb->request_bufflen = old_request_bufflen;
635     srb->use_sg = old_sg;
636     srb->sc_data_direction = old_sc_data_direction;
637     srb->cmd_len = old_cmd_len;
638     memcpy(srb->cmnd, old_cmnd, MAX_COMMAND_SIZE);
639
640     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
641         US_DEBUGP("-- auto-sense aborted\n");
642         srb->result = DID_ABORT << 16;
643         goto Handle_Errors;
644     }
645     if (temp_result != USB_STOR_TRANSPORT_GOOD) {
646         US_DEBUGP("-- auto-sense failure\n");
647
648         /* we skip the reset if this happens to be a
649          * multi-target device, since failure of an
650          * auto-sense is perfectly valid
651          */

```

```

652         srb->result = DID_ERROR << 16;
653         if (!(us->flags & US_FL_SCM_MULT_TARG))
654             goto Handle_Errors;
655         return;
656     }
657
658     US_DEBUGP("-- Result from auto-sense is %d\n", temp_result);
659     US_DEBUGP("-- code: 0x%x, key: 0x%x, ASC: 0x%x, ASCQ: 0x%x\n",
660         srb->sense_buffer[0],
661         srb->sense_buffer[2] & 0xf,
662         srb->sense_buffer[12],
663         srb->sense_buffer[13]);
664 #ifdef CONFIG_USB_STORAGE_DEBUG
665     usb_stor_show_sense(
666         srb->sense_buffer[2] & 0xf,
667         srb->sense_buffer[12],
668         srb->sense_buffer[13]);
669 #endif
670
671     /* set the result so the higher layers expect this data */
672     srb->result = SAM_STAT_CHECK_CONDITION;
673
674     /* If things are really okay, then let's show that. Zero
675      * out the sense buffer so the higher layers won't realize
676      * we did an unsolicited auto-sense. */
677     if (result == USB_STOR_TRANSPORT_GOOD &&
678         /* Filemark 0, ignore EOM, ILI 0, no sense */
679         (srb->sense_buffer[2] & 0xaf) == 0 &&
680         /* No ASC or ASCQ */
681         srb->sense_buffer[12] == 0 &&
682         srb->sense_buffer[13] == 0) {
683         srb->result = SAM_STAT_GOOD;
684         srb->sense_buffer[0] = 0x0;
685     }
686 }
687
688 /* Did we transfer less than the minimum amount required? */
689 if (srb->result == SAM_STAT_GOOD &&
690     srb->request_bufflen - srb->resid < srb->underflow)
691     srb->result = (DID_ERROR << 16) | (SUGGEST_RETRY << 24);
692
693 return;
694
695 /* Error and abort processing: try to resynchronize with the device
696  * by issuing a port reset. If that fails, try a class-specific
697  * device reset. */
698 Handle_Errors:
699
700     /* Set the RESETING bit, and clear the ABORTING bit so that
701      * the reset may proceed. */
702     scsi_lock(us_to_host(us));
703     set_bit(US_FLIDX_RESETTING, &us->flags);
704     clear_bit(US_FLIDX_ABORTING, &us->flags);
705     scsi_unlock(us_to_host(us));
706
707     /* We must release the device lock because the pre_reset routine
708      * will want to acquire it. */
709     mutex_unlock(&us->dev_mutex);
710     result = usb_stor_port_reset(us);

```

```

711     mutex_lock(&us->dev_mutex);
712
713     if (result < 0) {
714         scsi_lock(us_to_host(us));
715         usb_stor_report_device_reset(us);
716         scsi_unlock(us_to_host(us));
717         us->transport_reset(us);
718     }
719     clear_bit(US_FLIDX_RESETTING, &us->flags);
720 }

```

这段代码的复杂，调用关系一层又一层，让很多新手看了感觉无可奈何。

## 35. 迷雾重重的批量传输（二）

其实故事已经讲了很久，但如果你觉得到这里你已经把故事都看明白了，那么你错了。不仅仅是错了。不信，我们就继续看，先看 512 行，`us->transport()`，这个函数指针同样是在 `storage_probe` 时被赋值，对于 U 盘，它遵守的是 Bulk-Only 协议，因此 `us->transport()` 被赋值为 `usb_stor_Bulk_transport()`。来看 `usb_stor_Bulk_transport()`，它同样来自 `drivers/usb/storage/transport.c`：

```

941 int usb_stor_Bulk_transport(struct scsi_cmnd *srb, struct us_data *us)
942 {
943     struct bulk_cb_wrap *bcb = (struct bulk_cb_wrap *) us->iobuf;
944     struct bulk_cs_wrap *bcs = (struct bulk_cs_wrap *) us->iobuf;
945     unsigned int transfer_length = srb->request_bufflen;
946     unsigned int residue;
947     int result;
948     int fake_sense = 0;
949     unsigned int cswlen;
950     unsigned int cbwlen = US_BULK_CB_WRAP_LEN;
951
952     /* Take care of BULK32 devices; set extra Byte to 0 */
953     if (unlikely(us->flags & US_FL_BULK32)) {
954         cbwlen = 32;
955         us->iobuf[31] = 0;
956     }
957
958     /* set up the command wrapper */
959     bcb->Signature = cpu_to_le32(US_BULK_CB_SIGN);
960     bcb->DataTransferLength = cpu_to_le32(transfer_length);
961     bcb->Flags = srb->sc_data_direction == DMA_FROM_DEVICE ? 1 << 7 : 0;
962     bcb->Tag = ++us->tag;
963     bcb->Lun = srb->device->lun;
964     if (us->flags & US_FL_SCM_MULT_TARG)
965         bcb->Lun |= srb->device->id << 4;
966     bcb->Length = srb->cmd_len;
967
968     /* copy the command payload */
969     memset(bcb->CDB, 0, sizeof(bcb->CDB));
970     memcpy(bcb->CDB, srb->cmnd, bcb->Length);
971

```

```

972     /* send it to out endpoint */
973     US_DEBUGP("Bulk Command S 0x%x T 0x%x L %d F %d Trg %d LUN %d CL %d\n",
974               le32_to_cpu(bcb->Signature), bcb->Tag,
975               le32_to_cpu(bcb->DataTransferLength), bcb->Flags,
976               (bcb->Lun >> 4), (bcb->Lun & 0x0F),
977               bcb->Length);
978     result = usb_stor_bulk_transfer_buf(us, us->send_bulk_pipe,
979                                         bcb, cbwlen, NULL);
980     US_DEBUGP("Bulk command transfer result=%d\n", result);
981     if (result != USB_STOR_XFER_GOOD)
982         return USB_STOR_TRANSPORT_ERROR;
983
984     /* DATA STAGE */
985     /* send/receive data payload, if there is any */
986
987     /* Some USB-IDE converter chips need a 100us delay between the
988     * command phase and the data phase. Some devices need a little
989     * more than that, probably because of clock rate inaccuracies. */
990     if (unlikely(us->flags & US_FL_GO_SLOW))
991         udelay(125);
992
993     if (transfer_length) {
994         unsigned int pipe = srb->sc_data_direction == DMA_FROM_DEVICE ?
995                             us->recv_bulk_pipe : us->send_bulk_pipe;
996         result = usb_stor_bulk_transfer_sg(us, pipe,
997                                           srb->request_buffer, transfer_length,
998                                           srb->use_sg, &srb->resid);
999         US_DEBUGP("Bulk data transfer result 0x%x\n", result);
1000        if (result == USB_STOR_XFER_ERROR)
1001            return USB_STOR_TRANSPORT_ERROR;
1002
1003        /* If the device tried to send back more data than the
1004        * amount requested, the spec requires us to transfer
1005        * the CSW anyway. Since there's no point retrying the
1006        * the command, we'll return fake sense data indicating
1007        * Illegal Request, Invalid Field in CDB.
1008        */
1009        if (result == USB_STOR_XFER_LONG)
1010            fake_sense = 1;
1011    }
1012
1013    /* See flow chart on pg 15 of the Bulk Only Transport spec for
1014    * an explanation of how this code works.
1015    */
1016
1017    /* get CSW for device status */
1018    US_DEBUGP("Attempting to get CSW...\n");
1019    result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1020                                        bcs, US_BULK_CS_WRAP_LEN, &cswlen);
1021
1022    /* Some broken devices add unnecessary zero-length packets to the
1023    * end of their data transfers. Such packets show up as 0-length
1024    * CSWs. If we encounter such a thing, try to read the CSW again.
1025    */
1026    if (result == USB_STOR_XFER_SHORT && cswlen == 0) {
1027        US_DEBUGP("Received 0-length CSW; retrying...\n");
1028        result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1029                                            bcs, US_BULK_CS_WRAP_LEN, &cswlen);
1030    }

```



```

1031
1032     /* did the attempt to read the CSW fail? */
1033     if (result == USB_STOR_XFER_STALLED) {
1034
1035         /* get the status again */
1036         US_DEBUGP("Attempting to get CSW (2nd try)...\n");
1037         result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1038             bcs, US_BULK_CS_WRAP_LEN, NULL);
1039     }
1040
1041     /* if we still have a failure at this point, we're in trouble */
1042     US_DEBUGP("Bulk status result = %d\n", result);
1043     if (result != USB_STOR_XFER_GOOD)
1044         return USB_STOR_TRANSPORT_ERROR;
1045
1046     /* check bulk status */
1047     residue = le32_to_cpu(bcs->Residue);
1048     US_DEBUGP("Bulk Status S 0x%x T 0x%x R %u Stat 0x%x\n",
1049         le32_to_cpu(bcs->Signature), bcs->Tag,
1050         residue, bcs->Status);
1051     if (bcs->Tag != us->tag || bcs->Status > US_BULK_STAT_PHASE) {
1052         US_DEBUGP("Bulk logical error\n");
1053         return USB_STOR_TRANSPORT_ERROR;
1054     }
1055
1056     /* Some broken devices report odd signatures, so we do not check them
1057      * for validity against the spec. We store the first one we see,
1058      * and check subsequent transfers for validity against this signature.
1059      */
1060     if (!us->bcs_signature) {
1061         us->bcs_signature = bcs->Signature;
1062         if (us->bcs_signature != cpu_to_le32(US_BULK_CS_SIGN))
1063             US_DEBUGP("Learnt BCS signature 0x%08X\n",
1064                 le32_to_cpu(us->bcs_signature));
1065     } else if (bcs->Signature != us->bcs_signature) {
1066         US_DEBUGP("Signature mismatch: got %08X, expecting %08X\n",
1067             le32_to_cpu(bcs->Signature),
1068             le32_to_cpu(us->bcs_signature));
1069         return USB_STOR_TRANSPORT_ERROR;
1070     }
1071
1072     /* try to compute the actual residue, based on how much data
1073      * was really transferred and what the device tells us */
1074     if (residue) {
1075         if (!(us->flags & US_FL_IGNORE_RESIDUE)) {
1076             residue = min(residue, transfer_length);
1077             srb->resid = max(srb->resid, (int) residue);
1078         }
1079     }
1080
1081     /* based on the status code, we report good or bad */
1082     switch (bcs->Status) {
1083     case US_BULK_STAT_OK:
1084         /* device babbled -- return fake sense data */
1085         if (fake_sense) {
1086             memcpy(srb->sense_buffer,
1087                 usb_stor_sense_invalidCDB,
1088                 sizeof(usb_stor_sense_invalidCDB));
1089             return USB_STOR_TRANSPORT_NO_SENSE;

```

```

1090     }
1091
1092     /* command good -- note that data could be short */
1093     return USB_STOR_TRANSPORT_GOOD;
1094
1095     case US_BULK_STAT_FAIL:
1096         /* command failed */
1097         return USB_STOR_TRANSPORT_FAILED;
1098
1099     case US_BULK_STAT_PHASE:
1100         /* phase error -- note that a transport reset will be
1101          * invoked by the invoke_transport() function
1102          */
1103         return USB_STOR_TRANSPORT_ERROR;
1104     }
1105
1106     /* we should never get here, but if we do, we're in trouble */
1107     return USB_STOR_TRANSPORT_ERROR;
1108 }

```

这个函数也不是好“惹”的。但正是这个函数掀开了我们批量传输的新篇章。

## 36. 迷雾重重的批量传输（三）

在 `usb_stor_Bulk_transport()` 中，这个函数中调用的第一个最重要的函数，那就是 `usb_stor_bulk_transfer_buf()`。仍然是来自 `drivers/usb/stroage/transport.c`：

```

391 int usb_stor_bulk_transfer_buf(struct us_data *us, unsigned int pipe,
392     void *buf, unsigned int length, unsigned int *act_len)
393 {
394     int result;
395
396     US_DEBUGP("%s: xfer %u Bytes\n", __FUNCTION__, length);
397
398     /* fill and submit the URB */
399     usb_fill_bulk_urb(us->current_urb, us->pusb_dev, pipe, buf, length,
400         usb_stor_blocking_completion, NULL);
401     result = usb_stor_msg_common(us, 0);
402
403     /* store the actual length of the data transferred */
404     if (act_len)
405         *act_len = us->current_urb->actual_length;
406     return interpret_urb_result(us, pipe, length, result,
407         us->current_urb->actual_length);
408 }

```

如果结果提交成功了，那么返回值 `result` 将是 0。而 `act_len` 将记录实际传输的长度。不过光看这两个函数其实看不出什么，我们必须结合上下文来看。换句话说，我们需要结合 `usb_stor_Bulk_transport()` 中 `usb_stor_bulk_transfer_buf` 被调用的上下文，对比形参和实参来看，才能真的明白，才能拨开这团浓浓的迷雾。

而在仔细分析 `usb_stor_Bulk_transport()` 之前，我们先来看这个 `usb_fill_bulk_urb()` 函数。这个函数我们第一次见到，在 USB 世界里，和这个函数相似的函数还有 `usb_fill_control_urb()`，除此之外还有一个叫做 `usb_fill_int_urb()` 的函数，不用说，这几个函数是差不多的，只不过它们分别对应 USB 传输模式中的批量、控制和中断。唯一一处与 `usb_fill_control_urb` 不同的便是，批量传输不需要有一个 `setup_packet`。具体来看 `usb_fill_bulk_urb()` 定义于 `include/linux/usb.h`：

```
1207 static inline void usb_fill_bulk_urb (struct urb *urb,
1208                                     struct usb_device *dev,
1209                                     unsigned int pipe,
1210                                     void *transfer_buffer,
1211                                     int buffer_length,
1212                                     usb_complete_t complete_fn,
1213                                     void *context)
1214 {
1215     spin_lock_init(&urb->lock);
1216     urb->dev = dev;
1217     urb->pipe = pipe;
1218     urb->transfer_buffer = transfer_buffer;
1219     urb->transfer_buffer_length = buffer_length;
1220     urb->complete = complete_fn;
1221     urb->context = context;
1222 }
```

我们调用这个函数的目的是为了填充一个 `urb`，然后我们可以把这个 `urb` 提交给 USB Core 那一层。很显然，它就是为特定的管道填充一个 `urb`（最初 `urb` 申请时被初始化为 0 了）。

此处特意提一下 `usb_complete_t` 类型，在 `include/linux/usb.h` 中，有下面这一行。

```
961 typedef void (*usb_complete_t)(struct urb *);
```

这里用了 `typedef` 来简化声明，不熟悉 `typedef` 功能的人可以去查一下，`typedef` 的强大使得以下两种声明作用相同。

一种作用是：

```
void (*func1)(struct urb *);
void (*func2)(struct urb *);
void (*func3)(struct urb *);
```

另一种作用是：

```
typedef void (*usb_complete_t)(struct urb *);
usb_complete_t func1;
usb_complete_t func2;
usb_complete_t func3;
```

看出来了吧，如果要声明很多个函数指针，那么显然使用 `typedef` 一次，就可以一劳永逸了，以后声明就很简单了。所以，咱们也就知道，实际上 `urb` 中的 `complete` 是一个函数指针，它被设置为指向函数 `usb_stor_blocking_completion()`。不用说，这个函数之后肯定会被调用。

401 行, `usb_stor_msg_common()` 这个函数被调用, 它的作用是 `urb` 会被提交, 然后核心层去调度, 去执行它。

```

124 static int usb_stor_msg_common(struct us_data *us, int timeout)
125 {
126     struct completion urb_done;
127     long timeleft;
128     int status;
129
130     /* don't submit URBs during abort/disconnect processing */
131     if (us->flags & ABORTING_OR_DISCONNECTING)
132         return -EIO;
133
134     /* set up data structures for the wakeup system */
135     init_completion(&urb_done);
136
137     /* fill the common fields in the URB */
138     us->current_urb->context = &urb_done;
139     us->current_urb->actual_length = 0;
140     us->current_urb->error_count = 0;
141     us->current_urb->status = 0;
142
143     /* we assume that if transfer_buffer isn't us->iobuf then it
144      * hasn't been mapped for DMA. Yes, this is clunky, but it's
145      * easier than always having the caller tell us whether the
146      * transfer buffer has already been mapped. */
147     us->current_urb->transfer_flags = URB_NO_SETUP_DMA_MAP;
148     if (us->current_urb->transfer_buffer == us->iobuf)
149         us->current_urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
150     us->current_urb->transfer_dma = us->iobuf_dma;
151     us->current_urb->setup_dma = us->cr_dma;
152
153     /* submit the URB */
154     status = usb_submit_urb(us->current_urb, GFP_NOIO);
155     if (status) {
156         /* something went wrong */
157         return status;
158     }
159
160     /* since the URB has been submitted successfully, it's now okay
161      * to cancel it */
162     set_bit(US_FLIDX_URB_ACTIVE, &us->flags);
163
164     /* did an abort/disconnect occur during the submission? */
165     if (us->flags & ABORTING_OR_DISCONNECTING) {
166
167         /* cancel the URB, if it hasn't been cancelled already */
168         if (test_and_clear_bit(US_FLIDX_URB_ACTIVE, &us->flags)) {
169             US_DEBUGP("-- cancelling URB\n");
170             usb_unlink_urb(us->current_urb);
171         }
172     }
173
174     /* wait for the completion of the URB */
175     timeleft = wait_for_completion_interruptible_timeout(
176         &urb_done, timeout ? : MAX_SCHEDULE_TIMEOUT);
177
178     clear_bit(US_FLIDX_URB_ACTIVE, &us->flags);

```

```

179
180     if (timeleft <= 0) {
181         US_DEBUGP("%s -- cancelling URB\n",
182                 timeleft == 0 ? "Timeout" : "Signal");
183         usb_kill_urb(us->current_urb);
184     }
185
186     /* return the URB status */
187     return us->current_urb->status;
188 }

```

注意了，`usb_fill_bulk_urb` 这个函数只填充了 `urb` 中的几个元素，而 `struct urb` 里面包含了很多东西，不过有一些设置是共同的，所以才需要用 `usb_stor_msg_common()` 函数来设置，可以看出给这个函数传递的参数只有两个：一个就是 `us`，另一个是 `timeout`（在我们这个案例中传给它的值是 0），我们继续进入到这个函数中来把它看清楚、看明白。

首先看 131 行，让 `us->flags` 和 `ABORTING_OR_DISCONNECTING` 相与，`ABORTING_OR_DISCONNECTING` 宏定义于 `drivers/usb/storage/usb.h` 中：

```

70 /* Dynamic flag definitions: used in set_bit() etc. */
71 #define US_FLIDX_URB_ACTIVE 18 /* 0x00040000 current_urb is in use */
72 #define US_FLIDX_SG_ACTIVE 19 /* 0x00080000 current_sg is in use */
73 #define US_FLIDX_ABORTING 20 /* 0x00100000 abort is in progress */
74 #define US_FLIDX_DISCONNECTING 21 /* 0x00200000 disconnect in progress */
75 #define ABORTING_OR_DISCONNECTING ((1UL << US_FLIDX_ABORTING) | \
76                                  (1UL << US_FLIDX_DISCONNECTING))
77 #define US_FLIDX_RESETTING 22 /* 0x00400000 device reset in progress */
78 #define US_FLIDX_TIMED_OUT 23 /* 0x00800000 SCSI midlayer timed out */

```

它只是一个 flag，咱们知道，每一个 USB Mass Storage 设备，会有一个 `struct us_data` 的数据结构，即 `us`。所以，在整个 `probe` 的过程来看，它相当于一个“全局”变量，因此可以使用一些 `flags` 来标记一些事情。比如，此处对于提交 `urb` 的函数来说，显然它不希望设备此时已经处于放弃或者断开的状态，因为那样就没有必要提交 `urb` 了。

而下一个函数 `init_completion()`，只是一个队列操作函数，它被定义于 `include/linux/completion.h` 中：

```

39 static inline void init_completion(struct completion *x)
40 {
41     x->done = 0;
42     init_waitqueue_head(&x->wait);
43 }

```

它只是调用了 `init_waitqueue_head` 去初始化一个等待队列。而 `struct completion` 的定义也在同一个文件中：

```

13 struct completion {
14     unsigned int done;
15     wait_queue_head_t wait;
16 };

```

关于 `init_waitqueue_head` 咱们将在下面的故事中专门进行描述。

而接下来，都是在设置 `us` 的 `current_urb` 结构。看 147 行，`transfer_flags` 被设置成了 `URB_NO_SETUP_DMA_MAP`，而 `URB_NO_SETUP_DMA_MAP` 表明，如果使用 DMA 传输，则 `urb` 中 `setup_dma` 指针所指向的缓冲区是 DMA 缓冲区，而不是 `setup_packet` 所指向的缓冲区。不过这个 `setup_packet` 是控制传输特有的概念，对于批量传输，根本没有这个概念，所以我们完全可以不予理睬。

接下来 `URB_NO_TRANSFER_DMA_MAP` 则表明，如果 `urb` 有一个 DMA 缓冲区需要传输，则该缓冲区是 `transfer_dma` 指针所指向的那个缓冲区，而不是 `transfer_buffer` 指针所指向的那一个缓冲区。换句话说，如果没有设置这两个 DMA 的 flag，那么 USB Core 就会使用 `setup_packet`（仅对控制传输来说有意义）和 `transfer_buffer` 作为数据传输的缓冲区，然后下面两行就是把 `us` 的 `iobuf_dma` 和 `cr_dma` 赋给了 `urb` 的 `transfer_dma` 和 `setup_dma`。143 行到 146 行的注释表明，只要 `transfer_buffer` 被赋了值，那就假设有 DMA 缓冲区需要传输，于是就去设 `URB_NO_TRANSFER_DMA_MAP`。关于 DMA 这一段，因为比较难理解，所以我们多说几句。

首先，这里是两个 DMA 相关的 flag：`URB_NO_SETUP_DMA_MAP` 和 `URB_NO_TRANSFER_DMA_MAP`。注意这两个是不一样的，前一个是专门为控制传输准备的，因为只有控制传输需要有这个 Setup 阶段，需要准备一个 Setup packet。我们只看后一个，关于 `transfer_buffer` 和 `transfer_dma` 的关系，当初同样用下面的方法申请了 `us->iobuf` 的内存：

```
466     us->iobuf = usb_buffer_alloc(us->pusb_dev, US_IOBUF_SIZE,
467                                GFP_KERNEL, &us->iobuf_dma);
468     if (!us->iobuf) {
469         US_DEBUGP("I/O buffer allocation failed\n");
470         return -ENOMEM;
471     }
```

这里就有 `us->iobuf` 和 `us->iobuf_dma` 这两个东西，但是我们注意到，148 行和 149 行，在设置 `URB_NO_TRANSFER_DMA_MAP` 这个 flag 时，先做了一次判断，判断 `us->current_urb->transfer_buffer` 是否等于 `us->iobuf`，这是什么意思呢？我们在什么地方对 `transfer_buffer` 赋过值？答案是在 `usb_fill_bulk_urb` 中，我们把 `us->iobuf` 传递了过去，它被赋给了 `urb->transfer_buffer`，这样做就意味着我们这里将使用 DMA 传输，所以这里就设置了这个 flag。倘若我们不希望进行 DMA 传输，那很简单，我们在调用 `usb_stor_msg_common` 之前，不让 `urb->transfer_buffer` 指向 `us->iobuf` 就行了，反正这都是我们自己设置的。需要知道的是，`transfer_buffer` 是给各种传输方式中真正用来数据传输的，而 `setup_packet` 仅仅是在控制传输中发送 Setup 的包的，控制传输除了 Setup 阶段之外，也会有数据传输阶段，这一阶段要传输数据还是得靠 `transfer_buffer`，而如果使用 DMA 方式，那么就是使用 `transfer_dma`。

154 行，终于到了提交 `urb` 这一步了，`usb_submit_urb` 得到调用，作为 USB 设备驱动程序，我们不需要知道这个函数究竟在做什么，只要知道怎么使用就可以了。它的定义在

`drivers/usb/core/urb.c` 中，我们得知它有两个参数，一个是要提交的 `urb`，另一个是内存申请的 `flag`。这里我们使用的是 `GFP_NOIO`，意思就是不能在申请内存时进行 I/O 操作。道理很简单，这是一个存储设备，调用 `usb_submit_urb` 很可能是因为我们想要读些磁盘或者 U 盘，在这种情况下如果申请内存的函数又再一次去读写磁盘，那就有问题了，什么问题？嵌套。什么叫申请内存的函数也会读写磁盘？因为内存不够。使用磁盘作为交换分区不就方便了，所以申请内存时可能要的内存存在磁盘上，那就得交换回来。这不就读写磁盘了吗？所以我们为了读写硬盘而提交 `urb`，那么这个过程就不能再有 I/O 操作了，这样做的目的是为了杜绝出现嵌套死循环。

于是我们调用了 154 行就可以往下走，剩下的事情 USB Core 和 USB 主机去处理，至于这个函数本身的返回值，如果一切正常，`status` 将是 0。所以这里判断，如果 `status` 不为 0 那么就算出错了。162 行，一个 `urb` 被提交了之后，通常我们会把 `us->flags` 中置上一个 `flag`，`US_FLIDX_URB_ACTIVE`，让我们记录下这个 `urb` 的状态是活着的。

165 行，这里我们再次判断 `us->flags`，看是不是谁设置了 `aborting` 或者 `disconnected` 的 `flag`。稍后我们会看到谁会设置这些 `flag`，显然如果已经设置了这些 `flag` 的话，咱们就没必要往下了，这个 `urb` 可以取消了。

175 行，引出时间机制。

---

## 37. 迷雾重重的批量传输（四）

有时候我也被这个问题所困扰，我不知道是不明白，还是这世界变化太快。连 Linux 中都引入了过期这么一个概念。设置一个时间，如果时间到了该做的事情还没有做完，那么某些事情就会发生。

比如需要烤蛋糕，现在是 8 点 30，而我们要烤 45 分钟，所以希望闹钟 9 点一刻响，当时时间到了，闹钟就如期待的一样，响个不停。在计算机中，也需要做这样的事情，有些事情，需要时间控制，特别是网络、通信等，凡是涉及数据传输，就得考虑超时，换句话说，就是要定一个闹钟，你要是在这个给定的时间里还没做好你该做的事情，那么停下来，别做了，肯定有问题。比如，如果烤蛋糕烤了 45 分钟，发现蛋糕一点香味都没有，颜色也没变，那肯定有问题，别烤了，先检查一下烤箱是不是坏了，或者是不是停电了等。

而具体到这里，需要用一个闹钟，或者叫专业一点，定时器。如果时间到了，就执行某个函数，这个功能 Linux 内核的时间机制已经实现了，只需要按“说明书”调用相应的接口函数即可。看代码，175 行，`wait_for_completion_interruptible_timeout()` 函数被调用，`kernel/sched.c` 中定义了若干个这样的函数，我们不去看它们的定义，但是可以把它们的声明“贴”出来，来

自 `include/linux/completion.h`:

```
45 extern void FASTCALL(wait_for_completion(struct completion *));
46 extern int FASTCALL(wait_for_completion_interruptible(struct completion
*x));
47 extern unsigned long FASTCALL(wait_for_completion_timeout(struct completion
*x,
48                                     unsigned long timeout));
49 extern unsigned long FASTCALL(wait_for_completion_interruptible_timeout(
50                                     struct completion *x, unsigned long timeout));
```

很显然, `wait_for_completion` 是这一系列函数中最基本的, 其他几个函数都是基于它的扩展函数。与 `wait_for_completion` 对应的一个函数是 `complete()`。其用法和作用如下所示。

首先我们要用 `init_completion` 初始化一个 `struct completion` 的结构体变量, 然后调用 `wait_for_completion()`, 这样当前进程就会进入睡眠, 处于一种等待状态, 而另一个进程可能会去做某事。

当它做完了某件事情之后, 它会调用 `complete()` 函数, 一旦它调用这个 `complete()` 函数, 那么刚才睡眠的这个进程就会被唤醒。这样就实现了一种同步机制, 或者叫等待机制。

而我们现在用到的这个 `wait_for_completion_interruptible_timeout()` 函数则是在简单的同步机制上作了加强。它被唤醒有两种可能, 一种就是和之前一样, 由调用 `complete()` 函数的进程给唤醒, 或者, 就是设置一个闹钟, 时间一到闹钟就响了。

举例来说一下这个方案。比如小时候我如果第二天要参加考试, 那么前一天晚上要么跟我妈说好让她第二天早上记得叫我, 或者如果我妈不在家, 那我就定闹钟, 到时间了闹钟就把我唤醒。

总之, 这里设置了闹钟, 时间为 `MAX_SCHEDULE_TIMEOUT`, 这么做的道理就是希望别提交一个 `urb` 上去之后半天都没人理, 如果真的没人理说明出问题了, 别浪费时间了, 先撤销这个 `urb` 重新提交吧。所以我们看到 178 行就清除 `US_FLIDX_URB_ACTIVE` flag, 然后如果 `timeleft` 小于等于 0 就调用 `usb_kill_urb()` 函数撤销当前这个 `urb`。

那么除了这个闹钟以外, 这个同步机制在我们这个案例中又是怎么工作的呢? 别忘了刚才我们那句 `init_completion(&urb_done)`, `urb_done` 是一个 `struct completion` 结构体变量, 这个定义在 `usb_stor_msg_common()` 函数的第 1 行就出现了。显然 `completion` 是 Linux 中同步机制的一个很重要的结构体。同时我们又把 `&urb_done` 作为第 1 个参数传递给了 `wait_for_completion_interruptible_timeout()`。所以我们就等着看发送唤醒信号的 `complete()` 函数在哪里被调用的, 换句话说, 这里一旦睡去, 何时才能醒来。

还记得在调用 `usb_fill_bulk_urb()` 填充 `urb` 时设置了一个 `urb->complete` 指针吗? 没错, 当时咱们就看到了 `urb->complete=usb_stor_blocking_completion`, 这相当于向 USB 主机控制器驱动传达了一个信息。所以, 当 `urb` 传输完成了之后, USB 主机控制器会唤醒它, 但不会直接唤醒



它，而是通过执行之前设定的 `urb` 的 `complete()` 函数指针所指向的函数，即调用 `usb_stor_blocking_completion()` 函数去真正唤醒它。`usb_stor_blocking_completion()` 函数定义于 `drivers/usb/storage/transport.c` 中：

```
111 static void usb_stor_blocking_completion(struct urb *urb)
112 {
113     struct completion *urb_done_ptr = (struct completion *)urb->context;
114
115     complete(urb_done_ptr);
116 }
```

这个函数就两句话，但它调用了 `complete()` 函数，`urb_done_ptr` 就被赋为 `urb->context`，而 `urb->context` 是什么？`usb_stor_msg_common()` 函数中，138 行，可不就是把刚初始化好的 `urb_done` 赋给了它吗？很显然，这样做就是唤醒刚才睡眠的那个进程。换言之，到这里，`wait_for_completion()` 将醒来，从而继续往下走。

下面只剩下几行代码了。首先是 `clear_bit()` 清除 `US_FLIDX_URB_ACTIVE`，表明这个 `urb` 不再是活跃了。因为该干的事都干完了。如果是超时了，那么也是一样的，`urb` 都要被撤销了，当然就不用设为活跃了。最后一句，`usb_stor_msg_common()` 函数终于该返回了，`return us->current_urb->status`，返回的就是 `urb` 的“status”。于是我们总算可以离开这个函数了。也就是我们将回到 `usb_stor_bulk_transfer_buf()` 中来。剩下的几行代码无非是处理一下结果，我们暂且先不多说。

回过头来看 `usb_stor_Bulk_transport()` 函数，978 行，`usb_stor_bulk_transfer_buf()` 函数得到调用。第 1 个参数，`us`，无须多说。第 2 个参数，`us->send_bulk_pipe`，作为 U 盘来说，它除了有一个控制管道以外，还会有两个批量管道，一个是 IN，一个是 OUT。经历过此前的风风雨雨，我们已经对 USB 中那些名词不再神秘感，所谓管道无非就是一个 `unsigned int` 类型的数。`us->send_bulk_pipe` 和接下来我们立刻会遇到的 `us->recv_bulk_pipe` 都是在曾经那个令人回味的 `storage_probe()` 中调用 `get_pipes()` 函数获得的。然后第 3 个参数 `ccb`，下面看仔细了。

943 行，定义了这么一个指针 `ccb`，是 `struct bulk_cb_wrap` 结构体的指针，这是一个专门为 Bulk-only 协议特别准备的数据结构，来自 `drivers/usb/storage/transport.h`：

```
50 /* command block wrapper */
51 struct bulk_cb_wrap {
52     __le32  Signature;                /* contains 'USBC' */
53     __u32   Tag;                      /* unique per command id */
54     __le32  DataTransferLength;       /* size of data */
55     __u8    Flags;                    /* direction in bit 0 */
56     __u8    Lun;                      /* LUN normally 0 */
57     __u8    Length;                  /* of of the CDB */
58     __u8    CDB[16];                 /* max command */
59 };
```

在同一个文件中还定义了另一个数据结构：`struct bulk_cs_wrap`。

```
66 /* command status wrapper */
```

```
67 struct bulk_cs_wrap {
68     __le32  Signature;           /* should = 'USBS' */
69     __u32   Tag;                /* same as original command */
70     __le32  Residue;            /* amount not transferred */
71     __u8    Status;             /* see below */
72     __u8    Filler[18];
73 };
```

这两个数据结构对应于 CBW 和 CSW，即 Command Block Wrapper 和 Command Status Wrapper。事到如今，我们需要关注一下 USB Mass Storage Bulk-only Transport 协议了，因为 U 盘是按照这个协议规定的方式去传输数据的。Bulk-only 传输方式是首先由主机给设备发送一个 CBW，然后设备接收到了 CBW，它会进行解释，然后按照 CBW 中定义的那样去执行它该做的事情，然后它会给主机返回一个 CSW。

CBW 实际上是命令的封装包，而 CSW 实际上是状态的封装包。（命令执行后的状态，成功还是失败呢？所以需要使用这么一个状态包）。

这时候我们就可以查看 usb\_stor\_Bulk\_transport()函数中，调用 usb\_stor\_bulk\_transfer\_buf()之前的那几行究竟在干什么了。很明显，这些行都是在为 usb\_stor\_bulk\_transfer\_buf()这个函数调用做准备，真正精彩的部分还是在 usb\_stor\_bulk\_transfer\_buf()中。

943 行，struct bulk\_cb\_wrap \*bcb，赋值为(struct bulk\_cb\_wrap \*) us->iobuf。944 行，struct bulk\_cs\_wrap \*bcs，也赋值为(struct bulk\_cb\_wrap \*) us->iobuf，然后定义一个 unsigned int 的变量 transfer\_length，赋值为 srb->request\_bufflen。然后接下来就开始为 bcb 的各成员赋值了。如图 4.36.1 和图 4.36.2 所示，分别描述了 CBW 和 CSW 的数据格式。

Byte \ bit	7	6	5	4	3	2	1	0
0-3	dCBWSignature							
4-7	dCBWTag							
8-11	dCBWDataTransferLength							
12	bmCBWFlags							
13	Reserved (0)				bCBWLUN			
14	Reserved(0)			bCBWCBLength				
15-30	CBWCB							

图 4.36.1 CBW

Byte \ bit	7	6	5	4	3	2	1	0
0-3	dCBWSignature ("LaMS")							
4-7	dCBWTag							
8-11	dCSWDataResidue							
12	bCSWStatus							

图 4.36.2 CSW

959 行, `bcb->Signature=cpu_to_le32(US_BULK_CB_SIGN)`, `Signature` 对应 USB Mass Storage spec 中 CBW 的前 4 个 Bytes, 即 `dCBWSignature`, `US_BULK_CB_SIGN` 这个宏定义于 `drivers/usb/storage/transport.h` 中:

```
62 #define US_BULK_CB_SIGN          0x43425355      /*spells out USBC */
```

也不知道是谁规定的, 只有把 `dCBWSignature` 里边写上 `43425355h` 才能标志着个数据包是一个 CBW。另外, CBW 的传输全是遵守 Little Endian 的, 所以 `cpu_to_le32()` 这个宏需要使用, 来转换数据格式。

然后 `bcb->DataTransferLength` 对应 CBW 中的 `dCBWDataTransferLength`。这个就是标志 `ho` 主机希望这个端点传输多少个 Bytes 的数据。这里把 `cpu_to_le32(transfer_length)` 赋给了它。而 `transfer_length` 刚才已经说了, 就是 `srb->request_bufflen`。其实这几个变量名换来换去最重要记录的还是同一样东西。

`bcb->Flags`, 对应于 CBW 中的 `bmCBWFlags`, `bcb->Flags = srb->sc_data_direction == DMA_FROM_DEVICE ? 1 << 7 : 0`; 表明的是数据传输的方向, `DMA_FROM_DEVICE` 在前面讲过, 表示数据是从设备传向主存。而 `bmCBWFlags` 是 8 位的, 其中 `bit7` 表示方向; 0 表示 Data-Out, 即 from host to the device; 1 表示 Data-in, 即 from the device to the host。所以这里如果是 1 的话要左移 7 位。

`bcb->Tag = srb->serial_number`, 这个 Tag 对应 CBW 中的 `dCBWTag`, `dCBWTag` 的意义在于, 主机会 send 出去, 而设备将会把这个 Tag 的内容给打印出来。确切地说, 设备会回送一个 CSW 回来, 而在 CSW 中会有个 `dCSWTag`, 它的内容和这个 `dCBWTag` 是一样的, 所以实际上这就跟接头暗号似的。每一个 SCSI 命令都会被赋上一个 `serial_number`, 这里把它用在了 Tag 上。

`bcb->Lun = srb->device->lun`, 很简单, 对应 CBW 中的 `bCBWLUN`, 就是表示这个命令是发给哪个 LUN 的, 我们知道一个设备如果支持多个 LUN, 那么显然每个 LUN 会有一个编号。比如要读写 U 盘上的某个分区, 那么当然得指明是哪个分区了。如果设备不支持多个 LUN, 那么这儿会被设置为 0。不过需要注意, 这里 `bcb->Lun` 和 CBW 中的 `bCBWLUN` 并不完全对应,

bCBWLUN 只有 4 个 bit，而咱们这中定义时，LUN 是有 8 位的，低 4 位用来对应 bCBWLUN，而高 4 位实际上是用来表示 target id 的。所以接下来判断 us->flags 里边设了 US\_FL\_SCM\_MULT\_TARG 这个标志没有，如果有，说明是支持多个 target 的，于是就要记录下是哪个 target。

bc->Length = srb->cmd\_len，这个对应于 CBW 中的 bCBWCBLength，即命令的有效长度，单位是 Bytes。SCSI 命令的有效长度只能是 1 到 16 之间。接下来有个 CDB 数组，数组共 16 个元素，理由在前面讲 struct scsi\_cmnd 中的 cmd 就已经说过了。而 969 行，970 行正是把命令 srb->cmd 数组的内容复制至 bc->CDB 中。

这时候，usb\_stor\_bulk\_transfer\_buf 正式被调用了。传递给它的第三个参数正是 bc，而第四个参数是 US\_BULK\_CB\_WRAP\_LEN，它也定义于 drivers/usb/storage/transport.h 中：

```
61 #define US_BULK_CB_WRAP_LEN 31
```

31 就是 CBW 的长度，CBW 正是 31 个 Bytes。而 usb\_stor\_bulk\_transfer\_buf 无非就是提交一个 urb，然后就不用管事了，就等结果呗。而最终的结果是由 interpret\_urb\_result() 返回的，传输正确那么会返回 USB\_STOR\_XFER\_GOOD，而如果不正确，那么 usb\_stor\_Bulk\_transport() 中就直接返回了，返回值是 USB\_STOR\_TRANSPORT\_ERROR。如果正确，那么继续往下走，这才到真正的数据传输阶段。在真正开始将数据传输阶段之前，我们先来查看 interpret\_urb\_result() 函数。

## 38. 迷雾重重的批量传输（五）

在讲数据传输阶段之前，先解决刚才的历史遗留问题。usb\_stor\_bulk\_transfer\_buf() 中，406 行，有一个很有趣的函数 interpret\_urb\_result() 被调用。这个函数同样来自 drivers/usb/storage/transport.c 中：

```
265 static int interpret_urb_result(struct us_data *us, unsigned int pipe,
266                               unsigned int length, int result, unsigned int partial)
267 {
268     US_DEBUGP("Status code %d; transferred %u/%u\n",
269               result, partial, length);
270     switch (result) {
271
272         /* no error code; did we send all the data? */
273         case 0:
274             if (partial != length) {
275                 US_DEBUGP("-- short transfer\n");
276                 return USB_STOR_XFER_SHORT;
277             }
278     }
```

```

279         US_DEBUGP("-- transfer complete\n");
280         return USB_STOR_XFER_GOOD;
281
282     /* stalled */
283     case -EPIPE:
284         /* for control endpoints, (used by CB[I]) a stall indicates
285          * a failed command */
286         if (usb_pipecontrol(pipe)) {
287             US_DEBUGP("-- stall on control pipe\n");
288             return USB_STOR_XFER_STALLED;
289         }
290
291         /* for other sorts of endpoint, clear the stall */
292         US_DEBUGP("clearing endpoint halt for pipe 0x%x\n", pipe);
293         if (usb_stor_clear_halt(us, pipe) < 0)
294             return USB_STOR_XFER_ERROR;
295         return USB_STOR_XFER_STALLED;
296
297     /* babble - the device tried to send more than we wanted to read */
298     case -EOVERFLOW:
299         US_DEBUGP("-- babble\n");
300         return USB_STOR_XFER_LONG;
301
302     /* the transfer was cancelled by abort, disconnect, or timeout */
303     case -ECONNRESET:
304         US_DEBUGP("-- transfer cancelled\n");
305         return USB_STOR_XFER_ERROR;
306
307     /* short scatter-gather read transfer */
308     case -EREMOTEIO:
309         US_DEBUGP("-- short read transfer\n");
310         return USB_STOR_XFER_SHORT;
311
312     /* abort or disconnect in progress */
313     case -EIO:
314         US_DEBUGP("-- abort or disconnect in progress\n");
315         return USB_STOR_XFER_ERROR;
316
317     /* the catch-all error case */
318     default:
319         US_DEBUGP("-- unknown error\n");
320         return USB_STOR_XFER_ERROR;
321     }
322 }

```

应该说这个函数的作用是一目了然，就是根据传进来的参数 `result` 进行判断，从而采取相应的行动。`partial` 是实际传输的长度，而 `length` 是期望传输的长度，传输结束了当然要比这两者，因为有所期待，才会失望。`result` 是 `usb_stor_msg_common()` 函数的返回值，其实就是状态代码。如果为 0 说明一切都很顺利，结果也是成功的。268 这行，打印结果，同时打印出 `partial` 和 `length` 的比，注意两个 `%u` 中间那个 `/`，就是除号，或者说分割分子和分母的符号。

然后通过一个 `switch` 语句判断 `result`，为 0，说明至少数据有传输。然后有两种情况，于是返回不同的值，`USB_STOR_XFER_SHORT` 和 `USB_STOR_XFER_GOOD`。至于返回这些值之后会得到什么反应，让我们边走边看。目前只需要知道的是，对于真正传输完全令人满意的情况，

返回值只能是 USB\_STOR\_XFER\_GOOD。返回其他值都说明有问题。而这里作为传递给 switch 的 result，实际上是 USB Core 那一层传过来的值。

而我们注意到，interpret\_urb\_result 这个函数整个是被作为一个返回值出现在 usb\_stor\_bulk\_transfer\_buf()中的。换言之，前者返回之后，后者也马上就返回了，即再次返回到了 usb\_stor\_Bulk\_transport()中。因此，我们把视线拉回 usb\_stor\_Bulk\_transport()，981 行，如果 result 不为 USB\_STOR\_XFER\_GOOD，就说明有一些问题，于是索性 usb\_stor\_Bulk\_transport()也返回，没必要再进行下一阶段的传输了。否则，才可以进行下一阶段。

下一个阶段是所谓的 Bulk-only 传输，总共就是三个阶段，即命令传输阶段、数据传输阶段、状态传输阶段。很显然，真正最有意义的阶段就是数据传输阶段，而在此之前，我们已经讲了第一阶段，即命令传输阶段。下面我们可以来看一下数据阶段。

989 行,990 行,实在没话可说,某些公司的产品在命令阶段和数据阶段居然还得延时 100μs,最早时只发现了 Genesys Logic 公司的产品有这个问题,后来又发现更多的产品也有同样的问题。所以使用了 US\_FL\_GO\_SLOW 这个 flag,如果你有兴趣看一下早期的 Linux Kernel,你会发现那时候其实没有这个 flag,那时候定义了一个 USB\_VENDOR\_ID\_GENESYS,直接比较这个产品是不是 Genesys Logic 公司的,如果是的,那就考虑这个延时,否则就不用。

993 行, transfer\_length 可能为 0,因为有的命令并不需要您传输数据,所以它没有数据阶段。而对于那些有数据阶段的情况,我们进入 if 这一段。

994 行,没什么可说的,就是根据数据传输方向确定用接收管道还是发送管道。

然后,996 行,usb\_stor\_bulk\_transfer\_sg()函数真正地执行批量数据传输。这个函数来自 drivers/usb/storage/transport.c 中:

```

470 int usb_stor_bulk_transfer_sg(struct us_data* us, unsigned int pipe,
471                               void *buf, unsigned int length_left, int use_sg, int *residual)
472 {
473     int result;
474     unsigned int partial;
475
476     /* are we scatter-gathering? */
477     if (use_sg) {
478         /* use the usb core scatter-gather primitives */
479         result = usb_stor_bulk_transfer_sglist(us, pipe,
480                                                (struct scatterlist *) buf, use_sg,
481                                                length_left, &partial);
482         length_left -= partial;
483     } else {
484         /* no scatter-gather, just make the request */
485         result = usb_stor_bulk_transfer_buf(us, pipe, buf,
486                                             length_left, &partial);
487         length_left -= partial;
488     }

```

```

489
490     /* store the residual and return the error code */
491     if (residual)
492         *residual = length_left;
493     return result;
494 }

```

注释说得很清楚，这个函数是一个壳，真正干活的是它所调用或者说利用的那两个函数：`usb_stor_bulk_transfer_sglist()`和`usb_stor_bulk_transfer_buf()`。后者刚才已经遇到过了，而前者是专门为 scatter-gather 传输准备的函数，它也来自 `drivers/usb/storage/transport.c` 中：

```

416 static int usb_stor_bulk_transfer_sglist(struct us_data *us, unsigned int
pipe,
417         struct scatterlist *sg, int num_sg, unsigned int length,
418         unsigned int *act_len)
419 {
420     int result;
421
422     /* don't submit s-g requests during abort/disconnect processing */
423     if (us->flags & ABORTING_OR_DISCONNECTING)
424         return USB_STOR_XFER_ERROR;
425
426     /* initialize the scatter-gather request block */
427     US_DEBUGP("%s: xfer %u Bytes, %d entries\n", __FUNCTION__,
428             length, num_sg);
429     result = usb_sg_init(&us->current_sg, us->pusb_dev, pipe, 0,
430             sg, num_sg, length, GFP_NOIO);
431     if (result) {
432         US_DEBUGP("usb_sg_init returned %d\n", result);
433         return USB_STOR_XFER_ERROR;
434     }
435
436     /* since the block has been initialized successfully, it's now
437      * okay to cancel it */
438     set_bit(US_FLIDX_SG_ACTIVE, &us->flags);
439
440     /* did an abort/disconnect occur during the submission? */
441     if (us->flags & ABORTING_OR_DISCONNECTING) {
442
443         /* cancel the request, if it hasn't been cancelled already */
444         if (test_and_clear_bit(US_FLIDX_SG_ACTIVE, &us->flags)) {
445             US_DEBUGP("-- cancelling sg request\n");
446             usb_sg_cancel(&us->current_sg);
447         }
448     }
449
450     /* wait for the completion of the transfer */
451     usb_sg_wait(&us->current_sg);
452     clear_bit(US_FLIDX_SG_ACTIVE, &us->flags);
453
454     result = us->current_sg.status;
455     if (act_len)
456         *act_len = us->current_sg.Bytes;
457     return interpret_urb_result(us, pipe, length, result,
458             us->current_sg.Bytes);
459 }

```

在 `usb_stor_bulk_transfer_sg()` 函数中, 判断 `use_sg` 是否为 0, 从而确定是否用 `scatter-gather`。对于 `use_sg` 等于 0 的情况, 表示不用 `scatter-gather`, 那么调用 `usb_stor_bulk_transfer_buf()` 发送 `scsi` 命令。实际传递的数据长度用 `partial` 记录, 然后 `length_left` 就记录还剩下多少没传递, 初值当然就是期望传递的那个长度。每次减去实际传递的长度即可。对于 `use_sg` 不等于 0 的情况, `usb_stor_bulk_transfer_sglist()` 函数被调用。我们来看这个函数。

## 39. 迷雾重重的批量传输（六）

`usb_stor_bulk_transfer_sglist()` 函数有一定的“蛊惑性”, 我们前面说过, 之所以采用 `sglist`, 就是为了提高传输效率。我们更知道, `sg` 的目的就是让一堆不连续的 `buffers` 在一次 `DMA` 操作都传输出去。其实在 `USB` 的故事中, 事情并非如此。不过如果你对 `USB Core` 里边的行为不关心的话, 那就无所谓了。

423 行, 424 行, 退出或者断链接, 就不要传递数据。

然后 429 行, `usb_sg_init()` 函数被调用, 这个函数来自 `drivers/usb/core/message.c`, 也就是说, 它是 `USB` 核心层提供的函数, 干什么用的? 初始化 `sg` 请求。其第 1 个参数是 `struct usb_sg_request` 结构体的指针。这里我们传递了 `us->current_sg` 的地址给它, 这里 `us->current_sg` 第一次派上用场, 所以我们需要隆重介绍一下。

在 `struct us_data` 中, 定义了一个成员 `struct usb_sg_request current_sg`。曾几何时我们见到过 `current_urb`, 这里又来了一个 `current_sg`。也许你感觉很困惑。其实可以这样理解, 之前我们知道 `struct urb` 表示的是一个 `USB` 请求, 而这里 `struct usb_sg_request` 实际上表示一个 `scatter gather request`, 从我们非 `USB` 核心层的人来看, 这两个结构体的用法是一样的。对于每次 `urb` 请求, 我们所做的只是申请一个结构体变量或者说申请指针, 然后申请内存, 第二步就是提交 `urb`, 即调用 `usb_submit_urb()`, 剩下的事情 `USB Core` 就会去帮我们处理了, `Linux` 中的每个模块都给别人服务, 也同时享受着别人提供的服务。

你要想跟别人协同工作, 只要按照人家提供的函数去调用, 把你的指针你的变量传递给别人, 其他你根本不用管, 事成之后自然会通知你。同样对于 `sg request`, `USB Core` 也实现了这些, 我们只需要申请并初始化一个 `struct usb_sg_request` 的结构体, 然后提交, `USB Core` 自然就知道该怎么处理了。闲话少说, 先来看 `struct usb_sg_request` 结构体。它来自 `include/linux/usb.h`:

```
1350 struct usb_sg_request {
1351     int             status;
1352     size_t          Bytes;
1353
1354     /*
1355      * members below are private: to usbcore,
```



```

1356      * and are not provided for driver access!
1357      */
1358      spinlock_t          lock;
1359
1360      struct usb_device    *dev;
1361      int                  pipe;
1362      struct scatterlist    *sg;
1363      int                  nents;
1364
1365      int                  entries;
1366      struct urb            **urbs;
1367
1368      int                  count;
1369      struct completion     complete;
1370 };

```

整个 USB 系统都会使用这个数据结构，如果我们希望使用 scatter gather 方式的话。USB Core 已经为我们准备好了数据结构和相应的函数，我们只需要调用即可。一共有 3 个函数，它们分别是 `usb_sg_init`，`usb_sg_wait`，`usb_sg_cancel`。我们要提交一个 sg 请求，需要做的是先用 `usb_sg_init` 来初始化请求，然后 `usb_sg_wait()` 正式提交。如果想撤销一个 sg 请求，那么调用 `usb_sg_cancel` 即可。

我们虽说不用仔细去看着三个函数内部是如何实现的，但至少得知道该传递什么参数吧。不妨来仔细看一下 `usb_sg_init()` 被调用时传递给它的参数。第 1 个刚才已经说了，就是 sg request，第 2 个，需要告诉它是哪个 USB 设备要发送或接收数据，我们给它传递的是 `us->pusb_dev`，第 3 个，是哪个管道，这个没什么好说的，管道是上面一路传下来的。第 4 个参数，这是专门适用于中断传输的，被传输中断端点的轮询率，对于批量传输，直接忽略，所以我们传递了 0。第 5 个和第 6 个参数就分别是 sg 数组和 sg 数组中元素的个数。第 7 个参数，`length`，传递的就是我们希望传输的数据长度。最后一个是 slab flag，内存申请相关的一个 flag。如果驱动程序处于 block I/O 路径中应该使用 `GFP_NOIO`，这里 `SLAB_NOIO` 实际上是一个宏，实际上就是 `GFP_NOIO`。

为什么用 `SLAB_NOIO` 或者 `GFP_NOIO`，请参见前面章节讲到如何调用 `usb_submit_urb()`，理由当时就已经讲过了。这个函数成功返回值为 0，否则返回负的错误码。初始化好了之后就可以为 `us->flags` 设置 `US_FLIDX_SG_ACTIVE` 了，对这个 flag 陌生吗？还是回去看 `usb_submit_urb()`，当时我们也为 `urb` 设置了这么一个 flag，`US_FLDX_URB_ACTIVE`，其实历史总是惊人相似。当初我们对待 `urb` 的方式和如今对待 sg request 的方式几乎一样。所以其实是很好理解的。

对比一下当初调用 `usb_submit_urb()` 的代码，就会发现 441 行到 448 行这一段我们不会陌生，当年我们提交 `urb` 之前就有这么一段，在 `usb_stor_msg_common()` 函数中，只不过那时候是 `urb` 而不是 sg，这两段代码之间何其相似！只是年年岁岁花相似，岁岁年年人不同啊！451 行，`usb_sg_wait()` 函数得到调用。它所需要的参数就是 sg request 的地址，这里传递了 `us->current_sg` 的地址给它。这个函数结束，`US_FLIDX_SG_ACTIVE` 这个 flag 就可以 clear 掉了。返回值被保

存在 `us->current_sg.status` 中，然后把它赋给了 `result`。而 `us->current_sg.Bytes` 保存了实际传输的长度，把它赋给 `*act_len`，然后返回之前，再来一次调用 `interpret_urb_result()` 转换一下结果。

最后，`usb_stor_bulk_transfer_sg()` 函数返回之前还做了一件事，将剩下的长度赋值给了 `*residual`。`*residual` 是形参，实参是 `&srb->resid`。而最终 `usb_stor_bulk_transfer_sg()` 返回的值就是 `interpret_urb_result()` 翻译过来的值。但是需要明白的一点是，这个函数的返回就意味着 Bulk 传输中的关键阶段，即数据阶段的结束。剩下一个阶段就是状态阶段了，要传递的是 CSW，就像当初传递 CBW 一样。

回到 `usb_stor_Bulk_transport()` 函数中来，判断结果是否为 `USB_STOR_XFER_ERROR` 或者 `USB_STOR_XFER_LONG`，前者表示出错，而后者表示设备试图发送的数据比我们需要的数据要多。这种情况可以使用一个 `fake sense data` 来向上层汇报，出错了，但是和一般的出错不一样的是告诉上层，这个命令别再重发了。`fake_sense` 刚开始初始化为 0，这里设置为 1，后面将会用到。目前只需要知道的是，这种情况并不是不存在，实际上 USB Mass Storage Bulk-only spec 里边就定义了这种情况，spec 说了对这种情况，下一个阶段还是要照样进行。

最后，解释一点，`USB_STOR_XFER_LONG` 只是我们自己定义的一个宏，实际上是由 `interpret_urb_result()` 翻译过来的，真正从 USB Core 一层传递过来的结果是叫做 `-EOVERFLOW`，这一点在 `interpret_urb_result` 函数中能找到对应关系。`-EOVERFLOW` 顾名思义就是溢出。

最后，再解释一点。实际上 USB Core 这一层做的最人性化的一点就是对 `urb` 和对 `sg` 的处理了。写代码的人喜欢把数据传输具体化为 `request`，`urb` 和 `sg` 都被具体化为 `request`，即请求。而 USB Core 的能耐就是让写设备驱动的人能够只要申请一个请求，调用 USB Core 提供的函数进行初始化，然后调用 USB Core 提供的函数进行提交，这些步骤都是固定的，完全就像使用傻瓜照相机一样，然后进程可以睡眠，或者可以干别的事情，之后 USB Core 会通知你。你就可以接下来干别的事情了。

## 40. 迷雾重重的批量传输（七）

接下来我们该查看如何处理 CSW 了。1019 行，`usb_stor_bulk_transfer_buf()` 函数再一次被调用，这次是获得 CSW，期望长度是 `US_BULK_CS_WRAP_LEN`，这个宏来自 `drivers/usb/storage/transport.h` 中：

```
75 #define US_BULK_CS_WRAP_LEN 13
```

13 对应 CSW 的长度，13 个 Bytes。而 `cswlen` 记录了实际传输的长度。1026 行，如果返回值是 `USB_STOR_XFER_SHORT`，表明数据传少了，没有达到我们期望的那么多，而假如 `cswlen`

又等于 0，那么说明没有获得真正的 CSW，正如注释所说，有些设备会在数据阶段末尾多加一些 0 长度的包，这就意味着并没有获得 CSW，于是重新执行一次 `usb_stor_bulk_transfer_buf()`，再获得一次。

1033 行，如果 `result` 等于 `USB_STOR_XFER_STALLED`，在 `interpret_urb_result` 中查找一下，`USB_STOR_XFER_STALLED` 对应于 `usb core` 传回来的是 `-EPIPE`，这种情况说明管道不通，当然这也说明 `get CSW` 再次失败了，这种情况很简单，直接“retry”，为什么要“retry”？我们看一下 `interpret_urb_result()` 函数，最重要的就是 291 行到 295 行，这里判断了，因为我们曾经讲过，批量端点可能会设置了 `halt` 条件，设置了这种条件的端点必然会堵塞管道，所以这里就不管如何，试一试看，看清掉这个 `flag` 是否会有好转。所以对于这种情况，我们可以重试一次。抱着试一试的心态去“retry”，应该说这种心态是正确的。

这次传递给 `usb_stor_bulk_transfer_buf()` 函数的最后一个参数不是像之前的结果，这次是 `NULL`，这是因为实际上 `cswlen` 作为一个临时变量，表示的是状态阶段的实际传输长度，但是在眼下这种情况我们已经不需要使用这个临时变量了。

1043 行，如果都这么重新获取了还不成功的话，不用再耽误工夫了，直接返回吧，向领导汇报这设备无药可救了。没办法，返回 `USB_STOR_TRANSPORT_ERROR`，到这里还不成功那真的就是让人绝望了。

而从 1047 行开始，正式分析 CSW 了。结合图 4.13.5，那幅介绍 CSW 的格式的图，`bcs->Residue` 对应于 CSW 的 `dCSWDataResidue`，它表示实际传输的数据和期望传输的数据的差值。`bcs->Signature` 对应于 CSW 中的 `dCSWSignature`，`bcs->Tag` 对应于 CSW 中的 `dCSWTag`，而 `bcs->Status` 对应于 `bCSWStatus`。在 `bcs` 中成员的存储格式居然还有区别，有的是 Little Endian 的，有些却不是。对于那些 Little Endian 的，需要调用像 `cpu_to_le32` 这样的宏来转换，而其他的却不需要转换，对于 `bcs` 来说，其成员 `Residue` 和 `Signature` 就需要这样转换。这些规矩仿佛是没有道理的。

1051 行，和之前 `bcb` 中使用 `US_BULK_CB_SIGN` 一样，`US_BULK_CS_SIGN` 这个宏用来标志这个数据包是一个 CSW 包。而 `US_BULK_CS_OLYMPUS_SIGN` 也是一个宏，不过它是专为某种变态设备专门准备的。这两个宏和接下来将提到的一些宏依然来自 `drivers/usb/storage/transport.h` 中：

```
75 #define US_BULK_CS_WRAP_LEN 13
76 #define US_BULK_CS_SIGN          0x53425355      /* spells out 'USBS' */
77 #define US_BULK_STAT_OK          0
78 #define US_BULK_STAT_FAIL1
79 #define US_BULK_STAT_PHASE 2
```

对大多数普通的设备来说，如果要标志一个 CSW 包，其 `Signature` 会是 `53425355h`。但是 Olympus Camedia 这种数码相机偏偏要标新立异，愣是要换一个数字，那也没办法。

Tag 就是和 CBW 相对应的，两个 Tag 应该相同。要不然也就不叫接头暗号了。前面为 Tag 赋值为 srb->serial\_number，这回自然也应该等于这个值。

而 bcs->Status，标志命令执行是成功还是失败，当它是 0，表明命令是成功的，当它是非 0，肯定有问题。目前的 spec 规定，它只能是 00h, 01h, 02h，而 03h 到 FFh 都是保留的，不能用，所以这里会判断它是否是大于 US\_BULK\_STAT\_PHASE，也就是说是否会大于 02h，大于了当然就不行。也就是说这些条件如果不满足的话，那么一定是有问题的。返回错误值吧。

1074 行到 1079 行，如果 residue 不为 0，那么说明数据没传完，或者说和预期的不一样，那么来细看一下，首先该设备应该没有设置 US\_FL\_IGNORE\_RESIDUE 这个 flag，老规矩，让我们看一下什么样的设备设置了这个 flag：

```

520 /* Yakumo Mega Image 37
521 * Submitted by Stephan Fuhrmann <atomenergie@t-online.de> */
522 UNUSUAL_DEV( 0x052b, 0x1801, 0x0100, 0x0100,
523             "Tekom Technologies, Inc",
524             "300_CAMERA",
525             US_SC_DEVICE, US_PR_DEVICE, NULL,
526             US_FL_IGNORE_RESIDUE ),
527
528 /* Another Yakumo camera.
529 * Reported by Michele Alzetta <michele.alzetta@aliceposta.it> */
530 UNUSUAL_DEV( 0x052b, 0x1804, 0x0100, 0x0100,
531             "Tekom Technologies, Inc",
532             "300_CAMERA",
533             US_SC_DEVICE, US_PR_DEVICE, NULL,
534             US_FL_IGNORE_RESIDUE ),
535
536 /* Reported by Iacopo Spalletti <avvisi@spalletti.it> */
537 UNUSUAL_DEV( 0x052b, 0x1807, 0x0100, 0x0100,
538             "Tekom Technologies, Inc",
539             "300_CAMERA",
540             US_SC_DEVICE, US_PR_DEVICE, NULL,
541             US_FL_IGNORE_RESIDUE ),
542
543 /* Yakumo Mega Image 47
544 * Reported by Bjoern Paetzel <kolrabi@kolrabi.de> */
545 UNUSUAL_DEV( 0x052b, 0x1905, 0x0100, 0x0100,
546             "Tekom Technologies, Inc",
547             "400_CAMERA",
548             US_SC_DEVICE, US_PR_DEVICE, NULL,
549             US_FL_IGNORE_RESIDUE ),
550
551 /* Reported by Paul Ortyl <ortylp@3miasto.net>
552 * Note that it's similar to the device above, only different prodID */
553 UNUSUAL_DEV( 0x052b, 0x1911, 0x0100, 0x0100,
554             "Tekom Technologies, Inc",
555             "400_CAMERA",
556             US_SC_DEVICE, US_PR_DEVICE, NULL,
557             US_FL_IGNORE_RESIDUE ),

```

一般设备是不会设置这个 flag 的，但是确实有一些设备是设了这个 flag 的，查一查 drivers/usb/storage/unusual\_devs.h，发现 Tekom 公司的数码相机全都有这么一个问题。这个 flag

的意思很明确，对于这类设备不需要管 CSW 中的 `dCSWDataResidue`，因为十有八九这个字节汇报的东西是错的、不准的，当然这也就是有一个硬件 Bug 的例子了。

所以这里判断的就是这个 flag 没有设置，或者 `srb->sc_data_direction` 等于 `DMA_TO_DEVICE`，这种情况下，发送给设备的数据长度不应该超过 `transfer_length`。而 1077 行，`srb->resid` 本来是我们传递给 `usb_stor_bulk_transfer_sg` 的参数，记录的就是剩下的数据长度，（比如期待值是 10，传递了 8，那么剩余就是 2。），而 `residue` 刚刚被再次赋值了，不是原来的 `residue` 就是 `transfer_length`，要知道原来的 `residue` 等于 `dCSWDataResidue`，这是设备传递过来的，也就是硬件传来的数据，它未必就和我们软件得来的相同。所以 `srb->resid` 这时候就等于 `residue` 了，以硬件为准。

不过我想说的是，这几行代码实际上涉及一个鲜为人知的花絮。首先你看这段代码时，一定不明白为什么要判断传输方向是不是 `DMA_TO_DEVICE` 其实这里又是一个硬件的 Bug，开发设备驱动，最讲究的就是实战，尤其是像 `usb-storage` 这么一个通用的模块，它要支持各种各样的设备，不管你是三星的还是索尼的，只要你生产的是 `USB Mass Storage` 设备，而且你又不准备自己专门写一个设备驱动，那么这个 `usb-storage` 就应该支持你这个设备。

而在实战中我们发现，有些设备在执行读操作时，经常会在状态阶段汇报一个错误的 `dCSWDataResidue`，就是说比如本来没有传输完全顺利，该传几个字节就传了几个字节，按理说这种情况，`dCSWDataResidue` 应该记录 0，可是实际上这些设备却把这个值设成了某个正数，你说这不是胡来吗？而如果我们发现这个值是正数，那么当我们向 SCSI 核心层反映我们我们这个命令的结果时就会说这个命令执行失败了。

但事实上这些设备执行读操作并没有问题，这种情况属于谎报军情，罪该论斩。所以我们这里就加入这么一个标志，对于读操作，即方向为 `DMA_FROM_DEVICE` 的情况，就忽略这个 `dCSWDataResidue`，也就是忽略 `residue`，我们直接返回给 SCSI 的 `srb->resid` 就可以了。反之，对于写操作，即方向为 `DMA_TO_DEVICE` 的操作，我们当然不愿意无缘无故地抛弃有效的 `dCSWDataResidue`。

所以对于 `DMA_TO_DEVICE` 的情况，我们最终返回给 SCSI 核心层的 `srb->resid` 是以 `srb->resid` 和 `residue` 中那个大一点的为准。这就是为什么我们这里要判断或者我们设置了 `US_FL_IGNORE_RESIDUE` 这么一个 flag，或者我们执行的是读操作，对于这两种情况我们要忽略掉 `residue`，反之我们就不忽略。

不过有趣的是，大约五年前，某个外国人去开源社区抱怨，说他买了一个中国厂商生产的 MP3，该设备在写操作时总是莫名其妙地报错，但是在 Windows 下却用得好好的。最后大家一分析，发现问题就是在这里，即这个设备在写时会误报 `dCSWDataResidue`，用社区里面那些人的话说就是，这种设备在执行写命令时，会往 `dCSWDataResidue` 填写垃圾信息。本来写操作是正确执行了，可是偏偏要让 SCSI 那边以为操作没有执行成功。所以，某人就提交了一个 patch，

把这个判断方向的代码去掉了，因为反正读写都有可能出问题，那么干脆就不判断了，都给忽略掉得了，也因此，对于这种有问题的设备，就必须设置 `US_FL_IGNORE_RESIDUE` 这个 flag 了。当时那个 patch 是这样的：

```
==== drivers/usb/storage/transport.c 1.151 vs edited ====
--- 1.151/drivers/usb/storage/transport.c      2004-10-20 12:38:15 -04:00
+++ edited/drivers/usb/storage/transport.c      2004-10-28 10:50:42 -04:00
@@ -1058,8 +1058,7 @@
     /* try to compute the actual residue, based on how much data
      * was really transferred and what the device tells us */
     if (residue) {
-        if (!(us->flags & US_FL_IGNORE_RESIDUE) ||
-            srb->sc_data_direction == DMA_TO_DEVICE) {
+        if (!(us->flags & US_FL_IGNORE_RESIDUE)) {
             residue = min(residue, transfer_length);
             srb->resid = max(srb->resid, (int) residue);
         }
     }
```

同时我们也把当时去开源社区抱怨的那人的调试信息“贴”出来：

```
usb-storage: Command WRITE_10 (10 Bytes)
usb-storage: 2a 00 00 00 01 37 00 00 08 00
usb-storage: Bulk Command S 0x43425355 T 0x82 L 4096 F 0 Trg 0 LUN 0 CL 10
usb-storage: usb_stor_bulk_transfer_buf: xfer 31 Bytes
usb-storage: Status code 0; transferred 31/31
usb-storage: -- transfer complete
usb-storage: Bulk command transfer result=0
usb-storage: usb_stor_bulk_transfer_sglist: xfer 4096 Bytes, 2 entries
usb-storage: Status code 0; transferred 4096/4096
usb-storage: -- transfer complete
usb-storage: Bulk data transfer result 0x0
usb-storage: Attempting to get CSW...
usb-storage: usb_stor_bulk_transfer_buf: xfer 13 Bytes
usb-storage: Status code 0; transferred 13/13
usb-storage: -- transfer complete
usb-storage: Bulk status result = 0
usb-storage: Bulk Status S 0x53425355 T 0x82 R 3072 Stat 0x0
usb-storage: -- unexpectedly short transfer
usb-storage: scsi cmd done, result=0x10070000
SCSI error : <0 0 0 0> return code = 0x10070000
end_request: I/O error, dev sda, sector 311
```

应该说这段信息清晰地打印出来整个批量传输是怎么进行的，一共三个阶段：Command/Data/Status，这里执行的命令就是 `WRITE_10`，本来这是一次成功的传输，但是最后返回值 `result` 却不为 0，而是 `0x10070000`，关于这个 `0x10070000` 如何出来的，我们稍后会知道。最后两行是 SCSI Core 的代码打印出来的，我们不用管，只是需要知道我们最终返回给 scsi 核心层的一个有用信息就是 `srb->result`。所以我们看到 SCSI 那边打印了一个 `return code`，和我们这里的 `result` 是一样的。其实打印的都是 `srb->result`。很显然，`srb` 这个东西相当于 `usb-storage` 和 SCSI 那边的桥梁，连接了两个模块。

继续往下走。1081 行开始基于 CSW 返回的状态，来判断结果了。判断的值就是 `bcs->Status`。如果是 0，那么表明命令执行成功了。关于 `bcs->Status` 的取值，USB Mass Storage Bulk-only spec

里面规定得很清楚，如图 4.39.1 所示：

Value	Description
00h	Command Passed ("Good Status")
01h	Command Faild
02h	Phase Error
03h-04h	Reserved (Obsolete)
05h-FFh	Reserved

图 4.39.1 命令块状态值

我们前面看到，我们定义了三个宏 `US_BULK_STAT_OK`、`US_BULK_STAT_FAIL`、`US_BULK_STAT_PHASE`，分别对应 `00h`、`01h`、`02h`。所以这里我们就用这三个宏来进行判断。先来看后两个，如果是 `US_BULK_STAT_FAIL`，那么返回 `USB_STOR_TRANSPORT_FAILED`，如果是 `US_BULK_STAT_PHASE`，那么返回 `USB_STOR_TRANSPORT_ERROR`。这两个宏没什么好说的，我们在 `drivers/usb/storage/transport.h` 中一共定义了四个这样的宏：

```
99 #define USB_STOR_TRANSPORT_GOOD 0 /* Transport good, command good */
100 #define USB_STOR_TRANSPORT_FAILED 1 /* Transport good, command failed*/
101 #define USB_STOR_TRANSPORT_NO_SENSE 2 /* Command failed, no auto-sense*/
102 #define USB_STOR_TRANSPORT_ERROR 3 /* Transport bad (i.e. device dead)*/
```

其意思都很明显，从字面上就能看出来。至于这里返回这些值以后上面如何处理，那我们稍后从这个函数返回就知道了。另一个问题，`FAILED` 和 `ERROR` 的区别也很明显，一个是说传输没问题，但是命令执行时有错误，另一个是传输本身就有错。

现在我们来看一看 `US_BULK_STAT_OK` 的情况了，这种情况说明，传输成功了。但是这里需要判断 `fake_sense`。什么是 `fake_sense`？我们下一节再来专门讨论这个问题，需要知道的是，这里这三个 `return` 就意味着 `usb_stor_Bulk_transport()` 函数将结束了（当然，还有一个 `return`，1107 行，就是说如果以上情况都不属于，那当然更加是出错了，所以直接返回 `USB_STOR_TRANSPORT_ERROR`）。我们将返回 `usb_stor_invoke_transport()`，而这更加意味着一次批量传输的结束。至此我们就算是把这个迷雾重重的批量传输给从头到尾讲了一遍。而回到 `usb_stor_invoke_transport()` 之后所需要做的就是一些错误处理了，或者说秋后算账。

## 41. 跟着感觉走（一）

接下来的时间里我们会接触两个变量：`fake_sense` 和 `need_auto_sense`。`sense`，顾名思义就是感觉的意思。所以就让我们跟着感觉走。我们前面提到过，如果设备想发送比期望值更多的数据，那么前面就设了 `fake_sense` 为 1。这里就来查看设为 1 之后怎么办。这里我们看到了这个东西，`usb_stor_sense_invalidCDB`，它是谁？

让我们把镜头对准 drivers/usb/storage/scsiglue.c:

```
491 /* To Report "Illegal Request: Invalid Field in CDB */
492 unsigned char usb_stor_sense_invalidCDB[18] = {
493     [0]      = 0x70,                /* current error */
494     [2]      = ILLEGAL_REQUEST,    /* Illegal Request = 0x05 */
495     [7]      = 0x0a,                /* additional length */
496     [12]     = 0x24                  /* Invalid Field in CDB */
497 };
```

这是一个字符数组，共 18 个元素，初始化时其中 4 个元素被赋了值，为了说明这个数组，下面不得不插播一段 SCSI 广告，广告过后立刻回来。

我们知道 SCSI 通过命令通信，有一个命令是 Request Sense。它是用来获取错误信息的，不知道为什么，有人把错误信息称为“sense data”。如果一个设备接收到了一个 Request Sense 命令，那么它将按游戏规则返回一个 sense data，我们可以参考 SCSI 协议，找到 sense data 的格式规定，如图 4.40.1 所示。

Byte\Bit	7	6	5	4	3	2	1	0
0	Valid	Error code (70h or 71h)						
1	Segment number							
2	ilemark	EOM	ILI	Reserved	Sense key			
3	(MSB)  Information   (LSB)							
--								
6								
7	Additional sense length (n-7)							
8	(MSB)  Command-specific information   (LSB)							
--								
11								
12	Additional sense code							
13	Additional sense code qualifier							
14	Field replaceable unit code							
15	SKSV							
17								
18	Additional sense Bytes							
--								
n								

图 4.40.1 sense data 格式

标准的 sense data 是 18 个 Bytes 的。所以这里准备了一个 18 个元素的数组，第零个 Byte 的低 7 位称为 error code, 0x70 表明是出问题的是当前这个命令; 第二个 Byte 的低 4 位成为 sense key, 0x5h 称为 Illegal Request, 表明命令本身有问题，比如命令的参数不合法; 而第 7 个 Byte



称为 additional sense length 表明在这个 18 个元素之后还会有 additional sense Bytes，而它的长度就在这里被标注了，这些 additional sense Bytes 通常指的是一些命令特有的数据，或者是一些外围设备特有的数据，这里为它赋值为 0x0a。而第 12 个 Byte，称为 additional sense code，这部分针对 sense key 提供一些信息，也就是说比如 sense key 如果是 Illegal Request，那么我们知道命令有问题，那么究竟有什么问题呢？additional sense code 提供更详细的一些信息，SCSI 规范中对 24h 的描述是 Invalid Field in CDB，正是我们这里注释所说。

所以，这样我们明白了，1086 行，就是将 usb\_stor\_sense\_invalidCDB 数组里边的内容 copy 至 srb->sense\_buffer 里边，然后返回 USB\_STOR\_TRANSPORT\_NO\_SENSE。struct scsi\_cmnd 结构体里面是这样定义 sense\_buffer 的：

```
92 #define SCSI_SENSE_BUFFERSIZE 96
93     unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE];
94                                     /* obtained by REQUEST SENSE when
95                                     * CHECK CONDITION is received on original
96                                     * command (auto-sense) */
```

关于 sense\_buffer 就得从 SCSI 协议以及 Linux 中的 SCSI 核心层来讲了。SCSI 协议里有这么一码子事，当一个 SCSI 命令执行出了错，你可以发送一个 REQUEST\_SENSE 命令给目标设备，然后它会返回给你一些信息，即 sense data。不过，SCSI 核心层偷懒，把这一艰巨的任务抛给了底层驱动，即我们作为底层驱动不得不自己发送 REQUEST SENSE 命令给目标设备。

当然了，所谓 SCSI Core 偷懒并不是没有它的道理，因为有些 SCSI 主机卡会自动发送这个命令，就是说当设备汇报说命令执行有误，那这时 SCSI 主机卡会自动发送 REQUEST SENSE 命令去了解详情。所以 SCSI Core 就干脆把权力下放，让底层驱动自己去处理吧。因此稍后我们会看到一个变量名字叫做 need\_auto\_sense。就是说，REQUEST SENSE 这个命令要么就是硬件自动发出去，要么就让软件自动发出去，总之 SCSI Core 这一层是不管你了。只要你最终返回 SCSI 核心层时把相关的 sense data 保存在 srb->sense\_buffer 里，SCSI 核心层自然就知道该如何处理了。

再回到我们具体的问题中来，我们说了，有些设备你明明只期望它返回  $n$  个字节，它偏偏要给你捣乱，它想返回  $n+m$  个字节，对于这种情况我们怎么处理？老实说，它想多返回的几个字节我们完全可以抛弃，因为我们只关心我们提供的 buffer 是否装满了，是否达到了我们要求的 length 个字节。

如果达到了，那么剩下的不管也罢，不过写代码的同志们在这个问题上考虑得比我们要周到，他们对这个细节也是体贴入微的。或者说他们对这种设备也是很关心的。对于这种情况，写代码的同志们考虑，还是应该向上层汇报一下，说明这个命令对这个设备来说，执行起来总有一些问题。

因为这种情况完全可能就是，比如说，一个命令可以带有一些参数，而可能某个设备并不支持其中的某个参数，而你执行命令时去设置了这个参数，那么设备的返回值可能就不正常，

所谓的比预期值要多就是一种不正常的表现。所以呢，对于这种情况，我们干脆就告诉上层这个命令有问题，在 Linux 中，我们就可以通过 `sense data` 来向上层汇报。而之所以这里称作 `fake sense`，说的是这个 `sense data` 里面的东西是我们自己设置好的，因为我们已经很清楚我们应该在设备中放置什么，不需要向设备发送一个 `REQUEST SENSE` 的命令，而更确切地说，我们这个命令的返回结果是 `US_BULK_STAT_OK`，也就是说，从设备那方返回的状态来看，设备认为命令没有问题，但是你说设备的话你能相信吗？

老实说，从我进入复旦微电子系开始学习计算机硬件，我就没相信过硬件设备。不是这里有毛病就是那里有毛病，硬件 Bug 到处都是。继续说，因为设备认为传输是成功的，所以你发送 `REQUEST SENSE` 根本就没用，因为设备根本就不会为你准备 `sense data`，因为 `sense data` 本来就是为了提供错误信息的。因此我们需要自己设一个 `sense data`，放进 `sense_buffer` 里去，从而让 SCSI Core 那一层知道有这么一回事，别被设备瞒天过海给忽悠了。

讲到这里，`usb_stor_Bulk_transport()` 这个函数就算结束了。返回值一共就是四种情况，`USB_STOR_TRANSPORT_GOOD`，`USB_STOR_TRANSPORT_FAILED`，`USB_STOR_TRANSPORT_ERROR`，以及 `USB_STOR_TRANSPORT_NO_SENSE`。然后上层会去分析这些返回值。让我们结束这个函数，回到调用它的函数中来，即 `usb_stor_invoke_transport()`。在回去之前，我们需要记住的就是，对于刚才说的这种情况，即 `fake_sense` 为 1 的情况，我们返回的就是 `USB_STOR_TRANSPORT_NO_SENSE`。一会我们会看到 `usb_stor_invoke_transport()` 中是如何应付这种情况的。

## 42. 跟着感觉走（二）

回到 `usb_stor_invoke_transport()` 中来，517 行，还是老套路，又问是不是命令被放弃了，放弃了当然下面的就别执行了。

524 行，如果有错误，注意正如前面所说，`USB_STOR_TRANSPORT_ERROR` 表示传输本身就是有问题的，比如管道堵塞。而 `USB_STOR_TRANSPORT_FAILED` 则只是说明命令传输是没有问题的，就比如你作为场外观众给“非常 6+1”发短信了，然后李咏随机抽到你，给你打电话。电话通了，让你砸金蛋，砸出了金花你就能获得自己想要的奖品，但是问题是你没有砸中，就失去了机会，这属于 `FAILED`。但是另一种更惨的情况是，咏哥给你打电话还赶上你把手机关了，那你就只好认倒霉了，这就属于 `ERROR`。

对于这种疑似管道堵塞的问题，我们会调用自己写的一个函数 `us->transport_reset(us)`，`us->transport_reset()` 其实也是一个指针，我们也是很早以前和 `us->transport()` 一起赋的值，对于 U 盘，我们赋的值是 `usb_stor_Bulk_reset()`。所谓 `reset`，就相当于我们重启计算机，每次遇到些什

么乱七八糟的问题，我们二话不说，重启机器通常就会发现一切都好了。关于设备 reset，我们讲完命令的执行这一块之后再专门讲。对于这种情况，当然我们会设置 `srb->result` 为 `DID_ERROR`，然后返回。

531 行，看到了吧，这里就判断是不是 `USB_STOR_TRANSPORT_NO_SENSE` 了。如果是的，那么返回给 SCSI 的结果是 `SAM_STAT_CHECK_CONDITION`。返回这个值，SCSI 核心层那边就知道会去读 `srb->sense_buffer` 里边的东西。`SAM_STAT_CHECK_CONDITION` 是 SCSI 那边定义的宏，SCSI 协议规定，SCSI 总线上有若干个阶段，比如命令阶段、数据阶段、状态阶段，这三个阶段其实 Bulk-Only spec 里边也有。不过 SCSI 协议中还规定了更多的一些阶段，在 SCSI 协议中称一个阶段为一个 phase。除了这三个 phase 以外，还可以有 bus free phase、selection phase、message phase 等。而状态阶段就是要求目标设备返回给主机一个状态码（status code）。关于这些状态码，在 SCSI 的规范中定义得很清楚。在 `include/scsi/scsi.h` 中也有相关的宏定义。

```
125 /*
126  * SCSI Architecture Model (SAM) Status codes. Taken from SAM-3 draft
127  * T10/1561-D Revision 4 Draft dated 7th November 2002.
128  */
129 #define SAM_STAT_GOOD 0x00
130 #define SAM_STAT_CHECK_CONDITION 0x02
131 #define SAM_STAT_CONDITION_MET 0x04
132 #define SAM_STAT_BUSY 0x08
133 #define SAM_STAT_INTERMEDIATE 0x10
134 #define SAM_STAT_INTERMEDIATE_CONDITION_MET 0x14
135 #define SAM_STAT_RESERVATION_CONFLICT 0x18
136 #define SAM_STAT_COMMAND_TERMINATED 0x22 /* obsolete in SAM-3 */
137 #define SAM_STAT_TASK_SET_FULL 0x28
138 #define SAM_STAT_ACA_ACTIVE 0x30
139 #define SAM_STAT_TASK_ABORTED 0x40
```

其中，`SAM_STAT_CHECK_CONDITION` 就是对应 SCSI 协议中的 CHECK CONDITION，这 1 个状态表明有 sense data 被放置在相应的 buffer 里，于是 SCSI Core 那边就会去读 sense buffer。而我们这里遇到这种情况，当然就可以返回了。

536 行，要是没别的意外，到了这里我们就可以设置 `srb->result` 为 `SAM_STAT_GOOD` 了，说明一切都是好的。当然，对于之后会出现的 REQUEST SENSE 的执行失败，我们会再次修改 `srb->result` 的。

下面 543 行，出现了一个叫做 `need_auto_sense` 的变量，这是我们定义的临时变量，这里赋初值为 0。551 行到 565 行，两个 if 语句，为 `need_auto_sense` 赋了值，赋值为 1。我们首先很容易理解第 2 个 if，正如我们前面介绍的那样，REQUEST SENSE 这个艰巨的任务被下放给了底层驱动，那么我们就勇敢地去承担它，在 `USB_STOR_TRANSPORT_FAILED` 的情况下，我们就去发送一个 REQUEST SENSE 命令。设置了这个 flag 后就会看到我们会因此而执行 REQUEST SENSE。

那么第 1 个 if 呢？应该就不需要我解释了。对于那些遵守 CB 或者 DPCM 协议的设备它们自己没有办法决定状态，所以 SCSI 核心层当然就不知道去读它们的 sense buffer 了，但是不读 sense buffer 我们连这个命令执行成功与否都不知道，那怎么行？而设备对于大多数读操作的错误也不需要使用 sense buffer，因为它们对于读操作的错误通常会停止掉 Bulk-in 管道，这已经是一个很明显的信号了，不需要再检测 sense buffer。因为检测 sense buffer 或者说检测 sense data 的目的无非就是出错了，以后想知道出错的原因，而这种情况下原因已经清楚了。至于为什么读操作会有这种特点，那我只能说两个字，经验。写设备驱动靠的就是经验。

571 行，srb->resid 大于 0，说明有问题，希望传输  $n$  个字节，结果汇报上来说只传递了  $n-m$  个字节。对于这里列出的这五个命令，少传几个字节倒是无所谓。比如 INQUIRY，我就想知道设备的基本信息，那你说你姓甚名谁，生辰八字，学历如何，婚否等这些信息，你多说两句和少说两句，无所谓，没什么影响，但是有些命令就不能少传输了，比如我要传一个 pdf 文档，传到一半就不传了，那肯定不行，直接导致我可能打不开这个 pdf 文档。这里就是调用 US\_DEBUGP 打印一句调试语句，也就不退出了。

接下来的一些行，581 行开始，一直到 686 行，就是为 need\_auto\_sense 的情况发送 REQUEST SENSE 命令了。其实和之前一样，我们还是等于再进行一次批量传输，还是三个阶段，不过我们有了之前的经验，现在再看批量传输就简单多了，无非就是准备一个 struct scsi\_cmnd，调用 us->transport(us->srb, us)，然后结束了之后检查结果。这正是这一百多行代码所做的事情。不过我们偷懒了，没有另外申请一个 struct scsi\_cmnd，而是直接利用之前的 srb，只是调用 us->transport 之前先把一些关键的东西备份起来，然后执行完 us->transport 之后再恢复过来。所以接下来我们看到如下事件：

- 用 old\_cmnd 保存了 srb->cmnd;
- 用 old\_cmd\_len 保存了 srb->cmd\_len;
- 先把 srb->cmnd 清零，然后对它重新赋值，按 REQUEST SENSE 的意思来赋值。

不同的命令集里 REQUEST SENSE 的长度也不同，对于 RBC 或者咱们的 SCSI，长度为 6，而对于别的命令集，其长度为 12。

- 用 old\_sc\_data\_direction 保存了 srb->sc\_data\_direction，而把 srb->sc\_data\_direction 设置为 REQUEST SENSE 的要求，DMA\_FROM\_DEVICE。很显然，REQUEST SENSE 是向设备要 sense data，那么当然数据传输方向是从设备到主机。
- 用 old\_request\_buffer 来保存 srb->request\_buffer，而将 srb->request\_buffer 设置为 us->sensebuf，同时用 old\_request\_bufflen 来备份 srb->request\_bufflen，同时把 srb->request\_bufflen 设置为 18。
- 用 old\_sg 来备份 srb->use\_sg，而把 srb->use\_sg 设置为 0，传这么点儿数据就别用那麻烦的 scatter-gather 机制了。
- 用 old\_serial\_number 来备份 srb->serial\_number，并把 srb->serial\_number 的最高位取反。

- 用 `old_resid` 来备份 `srb->resid`，而把 `srb->resid` 再次初始化为 0。

这时候就可以调用 `us->transport(us->srb, us)` 了。并且用一个临时变量 `temp_result` 来保存这个结果。

这次命令完了之后，从 631 行到 638 行，就把刚才备份的那些变量给恢复原来的值。

640 行，再一次判断是不是被放弃了，如果是又 `goto Handle_Abort`。

645 行，然后判断 `temp_result`，如果这个 `result` 说明这次传输还有问题，那就说明连 `REQUEST SENSE` 都 `fail` 了。于是我们会设置 `srb->result=DID_ERROR<<16`。在这之前我们还会调用 `us->transport_reset(us)` 把设备 `reset`，因为连 `REQUEST SENSE` 都出错本身就说明很不正常。当然这里有一个条件，我们判断的是 `US_FL_SCM_MULT_TARG` 这个 `flag` 没有设置，因为设置了这个 `flag` 的设备有多个 `target`，这种情况下就不好胡乱全“`reset`”了，因为 `REQUEST SENSE` 这个命令虽然是一个基本的命令，但是毕竟执行成功与否无所谓，我们只是出于好奇才想知道一个命令执行出错的原因，即使不知道也没有太大的关系。没必要非得把一个多 `target` 的设备给“`reset`”了，不该管的事情不要管。

658 行到 669 行无非就是把 `temp_result` 的值打印出来，把 `sense_buffer` 里的值打印出来。这些都是调试信息。对调试设备驱动非常管用。

继续讲之前，我们先看一下 665 行，`usb_stor_show_sense()` 这个函数是第一次出现，曾经我们见过一个类似的函数，名叫 `usb_stor_show_command()`，它们都来自 `drivers/usb/storage/debug.c`：

```
161 void usb_stor_show_sense(
162     unsigned char key,
163     unsigned char asc,
164     unsigned char ascq) {
165
166     const char *what, *keystr;
167
168     keystr = scsi_sense_key_string(key);
169     what = scsi_extd_sense_format(asc, ascq);
170
171     if (keystr == NULL)
172         keystr = "(Unknown Key)";
173     if (what == NULL)
174         what = "(unknown ASC/ASCQ)";
175
176     US_DEBUGP("%s: ", keystr);
177     US_DEBUGPX(what, ascq);
178     US_DEBUGPX("\n");
179 }
```

这里面又调用了其他函数，`scsi_sense_key_string` 和 `scsi_extd_sense_format`。这两个函数来自 `driver/scsi/constants.c`。先来看对 `usb_stor_show_sense` 这个函数的调用。传递给它的实参是 `srb->sense_buffer` 中的几个元素，对比前面“贴”出来的那个 `sense data` 的格式，可知 `sense_buffer[2]`

的低 4 位被称为 Sense Key, 而 `sense_buffer[12]` 是 Additional sense code, 也称 ASC, `sense_buffer[13]` 是 Additional sense code qualifier, 也称 ASCQ。这三个元素联手为 mid level 提供了需要的信息, 主要也就是错误信息或者异常信息。

为什么要三个元素呢? 实际上就是一个分层的描述方法, 比如要描述某个房间就要说某城市某街道某门牌号。这三个元素也是起着这么一个作用。Sense Key 是第一层, ASC 则是对它的补充, 而 ASCQ 则又是对 ASC 的补充, 或者说解释。这样我们再来查看 `usb_stor_show_sense` 就很清楚了, 我们传递进来的是三个 char 变量, 而实际的信息就像某种编码一样被融入了这些 char 变量中, 而调用的两个来自 SCSI 核心层的函数 `scsi_sense_key_string` 和 `scsi_extd_sense_format` 就是起着翻译的作用, 也叫解码。解码了就可以打印出来了。

672 行, `srp->result` 设置为 `SAM_STAT_CHECK_CONDITION`。Request Sense 执行完之后, SCSI 规范告诉我们应该把 `srp->result` 设为 `SAM_STAT_CHECK_CONDITION`, 这样 mid level 就知道去检查 sense data。这也是为什么在 531 行、532 行会令 `srp->result` 也为这个值, 只不过那次 sense data 是我们自己手工准备的, 不是通过命令获得的。

677 行这个 if 这一小段, 首先我们需要明白, `need_auto_sense` 这个 flag 被设为 1 实际上是两种可能的, 它本身是在 `usb_stor_invoke_transport()` 中第 1 行所定义的一个局部变量, 并且在这个函数中特意把它初始化为 0。

第一处设置为 1 的位置是 551 行, 当时 `check_us->protocol` 为 `US_PR_CB` 或者 `US_PR_DPCM_USB`, 对于这种设备, (如果您只关心 U 盘, 那么就不理这种设备了。)

第二处设置这个 flag 的就是我们确实遇到了 failure, 562 行, `result` 如果等于 `USB_STOR_TRANSPORT_FAILED`, 这种情况当然要设置 `need_auto_sense` 了。而 704 行这里判断 `result` 是否等于 `USB_STOR_TRANSPORT_GOOD`, 那么很显然, 如果 `result` 等于 `USB_STOR_TRANSPORT_FAILED`, 那么它就不可能等于 `USB_STOR_TRANSPORT_GOOD`, 因此, 这里这个判断一定是针对第一种 `need_auto_sense` 的情况, 正如我们曾经说过的, 遵守 `US_PR_CB/US_PR_DPCM_USB` 协议的设备是不会自己返回命令执行之后的 Status, 所以我们不管它执行到底成功与否, 我们都会对它来一次 REQUEST SENSE, 就是为了尽可能多地获取一些信息, 这样一旦出了问题, 我们至少能多一些辅助信息来帮我们判断问题出在哪里。

那么对于 `USB_STOR_TRANSPORT_GOOD` 的情况, 首先这说明命令执行是没有问题的了, 我们仔细看一下这个 if 语句, 除了这个条件以外还判断了另外三个条件, (`srp->sense_buffer[2]&0xaf`) 结果为 0, 那么说明 `srp->sense_buffer[2]` 的 bit0~bit3 都为 0, bit5 为 0, bit7 也为 0, 而 bit4 和 bit6 是我们无所谓。虽然我们没有兴趣熟悉每一个 SCSI 命令的细节, 但应该有所了解, 所以让我们来仔细查看这个 `sense_buffer[2]`。

对照 sense data 的格式那张图, sense data 的第 2 个字节, bit0~bit3 是 sense key, bit4 是 Reserved, 即保留的, 不使用的。bit5 是 ILI, 全称 incorrect length indicator, bit6 是 EOM, 全

称 End of Medium, bit7 是 Filemark, 即卷标。

关于 sense key, Scsi 协议是这么规定的, 如果 sense key 为 0h, 那么这种情况表示 NO SENSE。这种情况通常对应于命令的成功执行或者就是 Filemark/EOM/ILI bits 中的任一个被设置为了 1。需要注意的是, SCSI 协议中边定义了四样东西, Filemark/EOM/ILI/Sense Key, 它们都是为了提供错误信息的, 只是前三者只要一个 bit 就能表达明确的意思了, 而最后一个包含很多信息, 所以需要用 4 个 bits, 并且还在后面附有很多额外信息, 即 sense\_buffer[12]和 sense\_buffer[13], 这里也要求它们为 0, 即所谓的 ASC 和 ASCQ 都为 0, 在 scsi 协议中面, 这种情况称之为 NO ADDITIONAL SENSE INFORMATION。

关于这一点 scsi 协议是这么说的: “The REQUEST SENSE command requests that the target transfer sense data to the initiator. If the target has no other sense data available to return, it shall return a sense key of NO SENSE and an additional sense code of NO ADDITIONAL SENSE INFORMATION.” 而这正是我们这里的代码所表达的意思。

(filemark 和 eom 都是针对磁带设备的, 跟磁盘设备无关。)

最后, 满足了这四个条件的情况就表示刚才这次 SCSI 命令的传输确实是圆满完成了。应该说这次检测还是蛮严格的。这里人家检查了这么多条件都满足然后就设置 `srp->result` 为 `SAM_STAT_GOOD`, 并且把 `srp->sense_buffer[0]`也置为 0。sense data 的 Byte 0 由两部分组成, Valid 和 Error code, 如果置为 0, 首先就说明这整个 sense data 是无效的, 用 scsi 标准的说法叫 invalid, 所以 SCSI Core 自然没法识别这么一个 sense data, 而我们既然认定这个命令是成功执行的, 当然就没有必然让 SCSI Core 再去理睬这么一个 sense data 了。

以上花了大量笔墨就讲了 677 行到 685 行这个 if 语句段。需要重新强调一点, 正如我们已经说过的, 对于 U 盘, 这段代码根本就不可能执行, 理由我们已经说过了。但是既然它出现在我们眼前了, 我们又有什么理由去逃避呢? 写代码, 尤其是写这种通用的设备驱动程序, 必然要考虑各种情况, 不是完全跟着感觉走, 也不是纯粹地追求华丽的算法和数据结构, 更应该接近实际。

这样, 关于 `need_auto_sense` 设置的这一段就结束了。最后还想重复一点, 说起来, REQUEST SENSE 这种命令应该由 mid level 来发, 不应该由底层驱动来发, 不过通常 mid-level 并不愿意发这个命令, 因为实际上很多 SCSI 主机适配卡 (SCSI host adapter) 会自动 “request the sense”。所以为了让事情变得简单, 设计上要求底层驱动去对付这个问题。所以要么 SCSI host adapters 自动获得 sense data, 要么就是 LLD (底层驱动程序) 去发送这个命令, 对于这个模拟的 SCSI 系统, 当然只能用软件去实现, 即必须在 LLD 中用代码来发送 request sense。

689 行, 如果经过了这么一番折腾, `srp->result` 仍然等于 `SAM_STAT_GOOD`, (我们在 536 行, 即进行 autosense 之前把 `srp->result` 设置成了 `SAM_STAT_GOOD`)。那么说明真金不怕火炼, 我们再判断最后一个条件, 即要求传输的数据长度是 `srp->request_bufflen`, 而实际上还剩下

`srb->resid` 个字节没有被传送，这种情况本身没什么，但是 `struct scsi_cmnd` 中有一个叫做 `underflow` 的成员，如果传输的数据连这个值都没有达到的话，不管其他条件如何，必须向上层反映出错了。

换句话说，有些 SCSI 命令有一个底线，你至少得达到这个底线！所以这里就是判断这么一个条件是否满足，如果传输的长度小于 `srb->underflow`，那么不用说，即便其他条件判断下来都觉得这个命令是成功的，我还是要汇报说你这个命令执行有误。而关于这种情况，我们反馈给 SCSI Core 的 `result` 是 `DID_ERROR<<16` 或上 `SUGGEST_RETRY<<24`。`DID_ERROR` 被定义为 `0x07`，`SUGGEST_RETRY` 为 `0x10`。其定义都在 `include/scsi/scsi.h` 中。所以这里 `srb->result` 就最终被设置为 `0x10070000` 了。

还记不记得当初我们“贴”出来的那个关于 `US_FL_IGNORE_RESIDUE` 关于 MP3 的调试信息了？回过去看一下，没错，当时的 `result` 就是 `0x10070000`，也就是这里赋的值。而当时之所以导致执行了这段代码，原因正是设备报虚警，明明读写正常，它偏要写一个“`residue`”到状态字节里去。导致代码在这里判断出读写出错。这个疑案只有到了这里我们才能真正明白。

至此，我们的这个故事也快接近尾声了。故事总有结束时，但 Linux 所反映的那种人们对自由的追求却是永无止境的。

693 行，`usb_stor_invoke_transport()` 函数终于返回了。如果之前有执行 `goto Handle_Errors`，那么 698 行会被执行，实际上就是设置 `srb->result` 为 `DID_ABORT<<16`，并且执行 `reset`。关于 `reset`，马上就会讲。

`usb_stor_invoke_transport()` 返回就回到了 `usb_stor_transparent_scsi_command()`，这个函数就是调用 `usb_stor_invoke_transport()`。

## 43. 有多少爱可以胡来？（一）

在驱动程序中，一个非常重要的概念就是错误处理。写代码不是写小说，不会因为作者的构思完美而天衣无缝。所以我们来查看在 `usb-storage` 中，是如何来进行错误处理的。

一切都得从结构体变量 `struct scsi_host_template usb_stor_host_template` 开始说起。其实这个结构体正是我们和 SCSI 核心层最关键的接口。我们知道这个结构体有很多内容，为 `usb_stor_host_template` 的许多成员赋了值，但是显然很多我们都没有讲。那么现在是时候去讲了。这其中，不妨把与错误处理相关的三个成员给揪出来。这些函数都是我们自己定义的，提供给 SCSI Core 那边去调用，就好像 `queuecommand()` 一样。

```
451      /* error and abort handlers */
```



```

452     .eh_abort_handler =      command_abort,
453     .eh_device_reset_handler = device_reset,
454     .eh_bus_reset_handler =  bus_reset,

```

好，让我们一个一个来看。先看两个与 reset 相关的函数，有人问我为什么有两个 reset 函数。很简单，当你的电脑有点小毛病了之后，可以有两种选择，一种是注销就可以了，一种是重启才可以。device\_reset 在这里对应的就是注销，bus\_reset 对应的就是重启。当然这样说并不严谨，只能说，一般来说轻微一点的问题 device\_reset 就够了，如果严重一点，眼看着设备病入膏肓了，那么可能就要 bus\_reset() 了。

我们让代码来告诉你。

首先看到的是 device\_reset()，来自 drivers/usb/storage/scsiglue.c 中：

```

279 /* This invokes the transport reset mechanism to reset the state of the
280  * device */
281 static int device_reset(struct scsi_cmnd *srb)
282 {
283     struct us_data *us = host_to_us(srb->device->host);
284     int result;
285
286     US_DEBUGP("%s called\n", __FUNCTION__);
287
288     /* lock the device pointers and do the reset */
289     mutex_lock(&(us->dev_mutex));
290     result = us->transport_reset(us);
291     mutex_unlock(&(us->dev_mutex));
292
293     return result < 0 ? FAILED : SUCCESS;
294 }

```

283 行，没什么好说的，us 还是那个 us。

调用 us->transport\_reset，我们前面曾经为 us->transport\_reset 赋值为 usb\_stor\_Bulk\_reset，所以这里也就是函数 usb\_stor\_Bulk\_reset() 会被调用，usb\_stor\_Bulk\_reset 定义于 drivers/usb/storage/transport.c 中：

```

1187 int usb_stor_Bulk_reset(struct us_data *us)
1188 {
1189     US_DEBUGP("%s called\n", __FUNCTION__);
1190
1191     return usb_stor_reset_common(us, US_BULK_RESET_REQUEST,
1192                                  USB_TYPE_CLASS | USB_RECIP_INTERFACE,
1193                                  0, us->ifnum, NULL, 0);
1194 }

```

进入这个函数一看，很简单，也不干别的，就是调用 usb\_stor\_reset\_common()。于是，我们接着来到了这个来自 driver/usb/storage/transport.c 中的 usb\_stor\_reset\_common() 函数。

```

1122 static int usb_stor_reset_common(struct us_data *us,
1123                                  u8 request, u8 requesttype,
1124                                  u16 value, u16 index, void *data, u16 size)

```

```

1125 {
1126     int result;
1127     int result2;
1128
1129     if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
1130         US_DEBUGP("No reset during disconnect\n");
1131         return -EIO;
1132     }
1133
1134     result = usb_stor_control_msg(us, us->send_ctrl_pipe,
1135                                   request, requesttype, value, index, data, size,
1136                                   5*HZ);
1137     if (result < 0) {
1138         US_DEBUGP("Soft reset failed: %d\n", result);
1139         return result;
1140     }
1141
1142     /* Give the device some time to recover from the reset,
1143      * but don't delay disconnect processing. */
1144     wait_event_interruptible_timeout(us->delay_wait,
1145                                     test_bit(US_FLIDX_DISCONNECTING, &us->flags),
1146                                     HZ*6);
1147     if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
1148         US_DEBUGP("Reset interrupted by disconnect\n");
1149         return -EIO;
1150     }
1151
1152     US_DEBUGP("Soft reset: clearing bulk-in endpoint halt\n");
1153     result = usb_stor_clear_halt(us, us->rcv_bulk_pipe);
1154
1155     US_DEBUGP("Soft reset: clearing bulk-out endpoint halt\n");
1156     result2 = usb_stor_clear_halt(us, us->send_bulk_pipe);
1157
1158     /* return a result code based on the result of the clear-halts */
1159     if (result >= 0)
1160         result = result2;
1161     if (result < 0)
1162         US_DEBUGP("Soft reset failed\n");
1163     else
1164         US_DEBUGP("Soft reset done\n");
1165     return result;
1166 }

```

前面几行是赋值，然后 `usb_stor_report_device_reset()` 被调用。`usb_stor_report_device_reset()` 定义于 `drivers/usb/storage/scsiglue.c` 中：

```

310 void usb_stor_report_device_reset(struct us_data *us)
311 {
312     int i;
313     struct Scsi_Host *host = us_to_host(us);
314
315     scsi_report_device_reset(host, 0, 0);
316     if (us->flags & US_FL_SCM_MULT_TARG) {
317         for (i = 1; i < host->max_id; ++i)
318             scsi_report_device_reset(host, 0, i);
319     }
320 }

```

315 行, `scsi_report_device_reset()`, `drivers/scsi/scsi_error.c` 中定义的。这个函数 SCSI Core 要求我们调用的。

Linux 世界里, 每一个人在乎的东西是不一样的。我们是不需要关心这个函数怎么定义的, 但是我们需要知道什么时候会调用它, 调用它干什么? 需要传递什么参数? 首先, 要传递三个参数, 第 1 个, `Scsi_Host` 指针, 一块 U 盘就有一个 `Scsi_Host`; 第 2 个参数, `channel`; 第 3 个参数 `target`, 描述 SCSI 设备位置的四个参数就三缺一了, 缺的就是 LUN, 因为一个设备都 `reset` 就不会管它上面有几个 LUN 了, 有几个都一起给它 `reset`。那么调用这个函数的目的告诉 SCSI 核心, 它观察到某个设备 `reset` 了。至于 SCSI 核心会如何处理呢, 那就不用管了。总而言之, 我们的职责是在发现了一个设备 `reset` 之后立刻向上级汇报。

`US_FL_SCM_MULT_TARG` 的这个 flag 也提过好几次, 它代表的是支持多个 `target`, 这是设备本身的属性, 不是咱们的代码给设备设置的。对于这种设备, `scsi_report_device_reset()` 就会被多调用几次, 针对每一个 `target` 要 `report` 一次。

结束了 `scsi_report_device_reset()`, 自然又回到了 `usb_stor_reset_common()`, 1123 行、1124 行设置一个 flag, 清除一个 flag, 设置的是 `US_FLIDX_RESETTING`, 清除的是 `US_FLIDX_ABORTING`, 关于这两个 flag, 下面会结合 `command_abort()` 来讲。

1131 行, `usb_stor_control_msg()` 被调用, 这个函数和我们调用的 `usb_stor_bulk_transfer_buf()` 差不多。它就是发送一个控制命令, 属于控制传输必备的函数, 其内部的工作很简单, 无非就是包装 `urb` 提交 `urb`, 然后判断结果。这里结合参数来查看传送的什么命令。首先, `us` 还是那个 `us`, 不再多说。然后, `pipe` 是 `us->send_ctrl_pipe`, 就是发送控制管道。`request`, `requesttype` 这些都是在调用 `usb_stor_reset_common()` 时传递进来的参数。在 `usb_stor_Bulk_reset()` 中可以看到, `request` 是 `US_BULK_RESET_REQUEST`, `requesttype` 是 `USB_TYPE_CLASS | USB_RECIP_INTERFACE`。

`US_BULK_RESET_REQUEST` 在 `drivers/usb/storage/transport.h` 中被设置为 `0xff`, 这是和 USB Mass Storage Class-Bulk Only transport 协议相对应的。该协议专门为 Bulk-Only Mass Storage 设备定义了一个请求, 即 `Reset`。协议中说: “this request is used to reset the mass storage device and its associated interface.” 协议中规定, 当 USB 主机要发送命令 `reset` USB 设备时, 需要通过发送控制管道发送一个请求, 即前面提过的 `ctrlrequest`, 其格式如图 4.42.1 所示。

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	11111111b	0000h	Interface	0000h	none

图 4.42.1 Bulk-Only Mass Storage Reset

其中 `bReques` 这一位须设置为 `255(FFh)`, `wValue` 设置为 `0`, `wIndex` 设置为接口序号, `wLength` 设置为 `0`。(而我们这里也确实这样做了, `wIndex` 被赋值为 `us->ifnum`, 和上次调用

usb\_stor\_control\_msg 时传递的一样，显然接口还是那个接口。

至于 requesttype，设置为 USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE，USB\_TYPE\_CLASS 定义于 include/linux/usb/ch9.h 中：

```
56 #define USB_TYPE_CLASS (0x01 << 5)
```

USB\_RECIP\_INTERFACE 也来源于同一个文件：

```
65 #define USB_RECIP_INTERFACE 0x01
```

这两个宏组合一下就是图 4.15.1 里 Reset 格式中的 bmRequestType。

这样，就完成了向设备发送 reset 命令的任务。返回值小于 0 就是出错了。

没出错那么就到 1141 行，wait\_event\_interruptible\_timeout()被调用。us->dev\_reset\_wait 前面讲过，它是一个等待队列头，在 storage\_probe()中被初始化。而以后在讲 storage\_disconnect()时会讲到，wake\_up(&us->dev\_reset\_wait)，它唤醒的正是这里进入睡眠的进程，这里 1141 行，会进入睡眠，进入睡眠之前先判断 US\_FLIDX\_DISCONNECTING 这个 flag 有没有设置，要是设了就没有必要睡眠了，直接退出。

1144 行，再判断一次，是真的设了这个 flag 那么直接“goto Done”，返回 rc，rc 就是初值 FAILED。返回之前先清除 US\_FLIDX\_RESETTING flag。关于 wait\_event\_interruptible\_timeout()这个函数，我们当初在分析 usb\_stor\_scan\_thread()时已经详细地讲过了，所以这里不需要再去多讲了。

当然，如果 US\_FLIDX\_DISCONNECTING 并没有设置，那么 6 秒时间到了，睡到自然醒，1150 行和 1153 行，usb\_stor\_clear\_halt()，有一种情况下要调用这个函数，即当设备 reset 之后，需要清除 halt 的这个 feature，然后端点才能正常工作。对于我们的两个批量端点，只要有一个清除 halt feature 失败了，那么整个负责 reset 的函数 usb\_stor\_reset\_common 就算失败了，并且因此会返回 FAILED，而且在返回之前先把 US\_FLIDX\_RESETTING 这个 flag 给去掉。

然后到了这里，usb\_stor\_reset\_commond 该返回，然后我们惊讶地发现，usb\_stor\_Bulk\_reset()也该返回，device\_reset()也该返回。就这样，我们走完 device\_reset()这么一个函数。之所以简单，是因为它的使命本身就很简单，其实就是给设备发送一个 reset 的请求，然后 clear 掉 halt feature，保证设备的端点没有停止。这就够了。

## 44. 有多少爱可以胡来？（二）

device\_reset()完了之后我们来看 bus\_reset()。同样来自 drivers/usb/storage/scsiglue.c 中：

```

296 /* Simulate a SCSI bus reset by resetting the device's USB port. */
297 static int bus_reset(struct scsi_cmnd *srb)
298 {
299     struct us_data *us = host_to_us(srb->device->host);
300     int result;
301
302     US_DEBUGP("%s called\n", __FUNCTION__);
303     result = usb_stor_port_reset(us);
304     return result < 0 ? FAILED : SUCCESS;
305 }

```

一点新鲜的东西都没有，就是调用 `usb_stor_port_reset()`，后者来自 `drivers/usb/storage/transport.c` 中：

```

1199 int usb_stor_port_reset(struct us_data *us)
1200 {
1201     int result, rc_lock;
1202
1203     result = rc_lock =
1204         usb_lock_device_for_reset(us->push_dev, us->push_intf);
1205     if (result < 0)
1206         US_DEBUGP("unable to lock device for reset: %d\n", result);
1207     else {
1208         /* Were we disconnected while waiting for the lock? */
1209         if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
1210             result = -EIO;
1211             US_DEBUGP("No reset during disconnect\n");
1212         } else {
1213             result = usb_reset_composite_device(
1214                 us->push_dev, us->push_intf);
1215             US_DEBUGP("usb_reset_composite_device returns %d\n",
1216                 result);
1217         }
1218         if (rc_lock)
1219             usb_unlock_device(us->push_dev);
1220     }
1221     return result;
1222 }

```

看完了 `device_reset` 再来看 `bus_reset` 应该就不难了。这个函数本身写得也特别清楚，注释很详细加上又有一些调试信息，基本上都应该能看懂。

咱们再简单介绍一下，和 `device_reset` 不同的地方在于，`device_reset` 实际上只是设备本身的复位，而 `bus_reset` 的意义就更加广泛了，它涉及从 USB Core 这一层面来初始化这个设备。千万不要混淆了，`bus reset` 不是说 USB bus 的 reset，而是 SCSI bus，即 SCSI 总线。哪来的 SCSI 总线？这是模拟的。难道你忘记了我们模拟一个 SCSI 主机、一个 SCSI 设备吗？实际上也就是模拟了一条 SCSI 总线。而这个东西的 reset 就意味着整个驱动都得重新初始化，即从 USB Core 开始，确切地说是 USB Hub 那边，先给我们在 USB 总线上来重新分配一个地址，重新建立起来之前的配置，重新选择 `altsetting`，也就是说对于我们这个设备，相当于重新经历了一次 USB 的总线枚举。而之后，`storage_probe()` 会重新被调用，整个模拟的 SCSI 总线将从头再来，所以说 `bus_reset` 是一场很大规模的运动。

283 行，还是老套路，检查是不是 disconnecting 了，要是的话就不用 reset 了。

286 行，还记得前面我曾说过一个接口对应一个驱动吗？一个设备可能有多个接口，也因此可能对应多个驱动，那么这里对一个设备 reset 就只能针对那种一个设备只有一个接口的情況，因为要不然就影响别人了，就好您想把自己建好的房子给拆了，那可能没人管，但假如您住的是楼房，您的卧室跟邻居家的卧室一墙之隔，那您要是敢把您那堵墙拆了，毋庸置疑，邻居可不答应。

然后 289 行到 300 行，总共三个来自 usb core 的关键函数：usb\_lock\_device\_for\_reset、usb\_reset\_device、usb\_unlock\_device。USB Core 都为您准备好了，要从总线上来“reset”设备，首先调用的是 usb\_lock\_device\_for\_reset，这是 USB 核心层的函数，来自 drivers/usb/core/usb.c 中，成功执行就返回 1 或者 0，失败了就返回一个负的错误代码。295 行的 usb\_reset\_device()，同样来自 USB 核心层 drivers/usb/core/hub.c 中，是 USB 核心层提供的用来重新初始化一个设备的函数。成功返回 0，否则就返回负的错误代码。出错了就得调用 usb\_unlock\_device 来释放锁。

最后 305 行，result 为 0 才是返回成功，否则返回 FAILED。

能看懂 296 行、297 行吗？我们说了，usb\_lock\_device\_for\_reset 可以返回负数，可以返回 1，可以返回 0。返回负数的情况就是 291 行到 293 行所做的事情，而对于 usb\_lock\_device\_for\_reset 来说，它返回 1 表示我们在执行了 usb\_reset\_device 之后要调用 usb\_unlock\_device 来释放锁，而返回 0 表示我们在执行了 usb\_reset\_device 之后不需要调用 usb\_unlock\_device 来释放锁。如果你对 these 很好奇，那么可以看一下 USB Core 的代码。特别是看一下关于两个宏的代码：USB\_INTERFACE\_BINDING 和 USB\_INTERFACE\_BOUND。

最后我们来看 command\_abort()函数，很显然这是一个错误处理函数，它的职责很明确，试图去中止当前的命令。同样来自 drivers/usb/storage/scsiglue.c 中：

```

244 /* Command timeout and abort */
245 static int command_abort(struct scsi_cmnd *srb)
246 {
247     struct us_data *us = host_to_us(srb->device->host);
248
249     US_DEBUGP("%s called\n", __FUNCTION__);
250
251     /* us->srb together with the TIMED_OUT, RESETING, and ABORTING
252      * bits are protected by the host lock. */
253     scsi_lock(us_to_host(us));
254
255     /* Is this command still active? */
256     if (us->srb != srb) {
257         scsi_unlock(us_to_host(us));
258         US_DEBUGP ("-- nothing to abort\n");
259         return FAILED;
260     }
261
262     /* Set the TIMED_OUT bit. Also set the ABORTING bit, but only if
263      * a device reset isn't already in progress (to avoid interfering

```

```

264     * with the reset). Note that we must retain the host lock while
265     * calling usb_stor_stop_transport(); otherwise it might interfere
266     * with an auto-reset that begins as soon as we release the lock. */
267     set_bit(US_FLIDX_TIMED_OUT, &us->flags);
268     if (!test_bit(US_FLIDX_RESETTING, &us->flags)) {
269         set_bit(US_FLIDX_ABORTING, &us->flags);
270         usb_stor_stop_transport(us);
271     }
272     scsi_unlock(us_to_host(us));
273
274     /* Wait for the aborted command to finish */
275     wait_for_completion(&us->notify);
276     return SUCCESS;
277 }

```

既然是阻止某个命令，那么传递进来的参数当然是 `struct scsi_cmnd` 的指针了。前两行的赋值就不多说了。

212 行，`us->srb` 表示当前的 `srb`，而 `srb` 是这个函数传进来的参数，`command_abort()` 希望中止当前的命令，而假如传进来的参数根本就不是当前的命令，那么肯定有问题。直接返回 `FAILED`。

然后，在 `abort` 的时刻，设置 `US_FLIDX_TIMED_OUT` 的 `flag`，然后如果设备没有 `reset`，那么再继续设置另一个 `flag`，`US_FLIDX_ABORTING`，接着调用 `usb_stor_stop_transport()`，这个函数的目的很简单，阻止任何进行中的传输，这个函数定义于 `drivers/usb/storage/transport.c` 中：

```

722 /* Stop the current URB transfer */
723 void usb_stor_stop_transport(struct us_data *us)
724 {
725     US_DEBUGP("%s called\n", __FUNCTION__);
726
727     /* If the state machine is blocked waiting for an URB,
728      * let's wake it up. The test_and_clear_bit() call
729      * guarantees that if a URB has just been submitted,
730      * it won't be cancelled more than once. */
731     if (test_and_clear_bit(US_FLIDX_URB_ACTIVE, &us->flags)) {
732         US_DEBUGP("-- cancelling URB\n");
733         usb_unlink_urb(us->current_urb);
734     }
735
736     /* If we are waiting for a scatter-gather operation, cancel it. */
737     if (test_and_clear_bit(US_FLIDX_SG_ACTIVE, &us->flags)) {
738         US_DEBUGP("-- cancelling sg request\n");
739         usb_sg_cancel(&us->current_sg);
740     }
741 }

```

这里有两个 `flag`，`US_FLIDX_URB_ACTIVE` 和 `US_FLIDX_SG_ACTIVE`，前一个 `flag` 此前讲 `usb_stor_msg_common()` 函数时，在 `urb` 被成功提交了之后，调用 `set_bit()` 宏为 `us->flags` 设置了这一位，于是这里就可以先判断，如果确实设置了，那么此时就可以调用 `usb_unlink_urb` 来从 `urb` 队列中取下来，因为没有必要再连接在上面了。

而 `US_FLIDX_SG_ACTIVE` 这个 flag 我们也不陌生，在批量传输里会用到。设置这个 flag 的是批量传输中的函数 `usb_stor_bulk_transfer_sglist()`，它会调用 `set_bit` 函数来设置这个 flag。于是这里也判断，如果确实设置了，那么此时就可以调用 `usb_sg_cancel` 来把 sg 取消。

看得出，`usb_stor_stop_transport()`的决心很坚定，目标很简单，就是要让传输进行不下去，有 urb 就取消 urb，有 sg 就取消 sg。

这时，在 230 行，咱们看到了等待函数，`wait_for_completion(&us->notify)`，`command_abort()` 函数进入等待。`usb_stor_control_thread()`中的 `complete(&(us->notify))`唤醒它。

第一种情况，您这里睡眠中，USB 设备断开了，于是 `usb_stor_control_thread` 要退出来，于是它会调用 `complete_and_exit()`来唤醒 `command_abort()`并且退出 `usb_stor_control_thread()`，这样一来，的话世界从此就清静了。

第二种情况，一个 SCSI 命令还没有开始执行，或者说即将要开始执行时，我们调用了 `command_abort()`，那么对应于 `usb_stor_control_thread()`中的 322 行到 325 行那几行的 if 语句，发现 `US_FLIDX_TIMED_OUT` flag 被设置了，于是，命令也不执行了，直接 `goto SkipForAbort`，然后调用 `complete(&(us->notify))`唤醒进入睡眠的进程。同时，对于 `usb_stor_control_thread()`来说，将 `us->srb` 设置为 NULL 就一切好了，可以开始下一轮循环了。

第三种情况，一个 SCSI 命令执行完了之后，才执行 `command_abort`，那么在这种情况下，由于设置了 `US_FLIDX_TIMED_OUT` 标志，对应于 `usb_stor_control_thread()`那边的 396 行、397 行的 if 语句，也没什么特别的。同样，执行 `complete(&(us->notify))`唤醒这个进入睡眠的进程，并且将 `us->srb` 设置为 NULL。

最后，234 行、235 行，清除这两个 `US_FLIDX_ABORTING` 和 `US_FLIDX_TIMED_OUT` 这两个 flag，接下来的传输还是照旧。而 `command_abort()`函数本身当然返回 `SUCCESS`，阻止了别人，它就成功了，所以它返回 `SUCCESS`。

至此我们就算讲完这三个负责错误处理的函数。那么关于这两个 flag 呢。实际上搜索一下整个内核目录，只有 `command_abort()`函数中设置了这两个 flag，而如果 `command_abort()`执行成功了，会在返回之前，把两个 flag 给清除掉。而需要测试 `US_FLIDX_TIMED_OUT` 这个 flag 的地方，仅仅只有 `usb_stor_control_thread()`和 `usb_stor_invoke_transport()`中。

关于前者正是我们刚才所讲的那样，除了唤醒 `command_abort()`以外，就是设置 `us->srb` 为 NULL，但总的来说，基本上对 SCSI 那边没有什么影响。而在 `usb_stor_invoke_transport()`中，如果我们遇到了 `US_FLIDX_TIMED_OUT` 被设置，那么情况会有所不同，因为这时候是在与 SCSI 层打交道，我们是因为要执行一个 SCSI 命令才进入到 `usb_stor_invoke_transport()`的，所以在这种情况下如果遇到了 abort，那么我们至少得给 SCSI 一个交代，正如我们在 `usb_stor_invoke_transport()`中第 724 行那一小段看到的那样，我们回复给 SCSI 层的是 `srb->result`，



我们把它设成了 `DID_ABORT<<16`。同时，对于 Bulk-only 的设备，我们还需要把我们的设备进行 reset。这是 Bulk-only 的传输协议中边规定的，即，在一个 abort 之后，必须要进行一次 reset，要不设备下次不能工作。（参见 Bulk-only transport 5.3.3.1 Phase Error/5.3.4 Reset Recovery）

再来看另一个 flag，`US_FLIDX_ABORTING`。实际上没有人专门去测试这个 flag 有没有设置，真正被测试的是另一个 flag，`ABORTING_OR_DISCONNECTING`，关于这个 flag 的定义我们早年曾经列出来过，实际上意思很明显，或者是设置了 Aborting 的 flag，或者就是设置了 Disconnecting 的 flag。

很多情况下我们实际上要判断的就是这两者之中任何一个是否被设置了，因为很显然，很多情况下，如果这两个中的任何一个被设置了，那么我们的某些事情就没有必要继续了，关于 `ABORTING_OR_DISCONNECTING`，一共有两个函数中需要判断它，一个是 `usb_stor_msg_common()`，另一个是 `usb_stor_bulk_transfer_sglist()`。其实，仔细回顾一下这两个函数，你就不难发现，在 `usb_stor_msg_common()` 中，其实我们就是在提交一个 urb 之前先检查这个 flag 有没有设置，提交了之后再次检查一下，实际上就相当于给了两次机会，因为提交了之后就相当于表白了之后，但是双方可能还没有发生什么实质性的故事，在这一刻如果你意识到这是一段没有结果的恋情，你还可以选择放弃。但是过了这第二次判断，那么结果就将是生米煮成熟饭了。而在 `usb_stor_bulk_transfer_sglist()` 中，其实道理是一样的，就是把 urb 的概念换成 sg，仅此而已。

到现在我们可以来看最后一个重量级的 flag 了。那就是 `US_FLIDX_DISCONNECTING`。而将带我们走入我们整个故事的最后一个重量级函数，`storage_disconnect()`。而这个函数也将宣告我们整个故事的结束，毕竟天下无不散的宴席，从我开始写这个故事时，我就知道一切都已注定，注定了开始，注定了结束。

---

## 45. 当梦醒了天晴了

我们来到了最后一个重要的函数，`storage_disconnect`。

USB 设备的热插拔特性注定了我们应该在设备插入时和拔出时做一些事情。主机和 USB 设备的“暧昧”关系体现在，需要它时，如胶似漆。但是，有一天 USB 设备必定要离开主机。对于主机来说，人生没有 USB 设备并不会不同。而且，事实上，USB 的即插即用特性也让主机知道，USB 设备并不曾真的离去，它们还会再相逢。前面见面时调用了 `storage_probe` 函数来让彼此接受对方，现在就该调用 `storage_disconnect` 函数来分手。而 USB 设备离开主机也需要处理一些后事。

相比 probe，disconnect 函数就简单多了，storage\_disconnect() 函数定义在 drivers/usb/storage/usb.c 中，这个函数不长。

```

1041 /* Handle a disconnect event from the USB core */
1042 static void storage_disconnect(struct usb_interface *intf)
1043 {
1044     struct us_data *us = usb_get_intfdata(intf);
1045     US_DEBUGP("storage_disconnect() called\n");
1046     quiesce_and_remove_host(us);
1047     release_everything(us);
1048 }

```

1044 行，虽然 usb\_get\_intfdata() 函数的确是第一次露面，但是这里的含义已然是司马昭之心路人皆知。没有讲过 usb\_get\_intfdata()，但是我们讲过 usb\_set\_intfdata()。前面讲 associate\_dev() 时，调用 usb\_set\_intfdata(intf,us)，当时分析了，这样做的结果就是使得 &intf->dev->driver\_data=us，而现在调用 usb\_get\_intfdata(intf) 的作用就是把 us 从中取出来，赋给这里的临时指针 us。

至于 quiesce\_and\_remove\_host() 和 release\_everything()，它们也都来自 drivers/usb/storage/usb.c 中：

```

859 /* First stage of disconnect processing: stop all commands and remove
860  * the host */
861 static void quiesce_and_remove_host(struct us_data *us)
862 {
863     struct Scsi_Host *host = us_to_host(us);
864
865     /* Prevent new USB transfers, stop the current command, and
866      * interrupt a SCSI-scan or device-reset delay */
867     scsi_lock(host);
868     set_bit(US_FLIDX_DISCONNECTING, &us->flags);
869     scsi_unlock(host);
870     usb_stor_stop_transport(us);
871     wake_up(&us->delay_wait);
872
873     /* It doesn't matter if the SCSI-scanning thread is still running.
874      * The thread will exit when it sees the DISCONNECTING flag. */
875
876     /* queuecommand won't accept any new commands and the control
877      * thread won't execute a previously-queued command. If there
878      * is such a command pending, complete it with an error. */
879     mutex_lock(&us->dev_mutex);
880     if (us->srb) {
881         us->srb->result = DID_NO_CONNECT << 16;
882         scsi_lock(host);
883         us->srb->scsi_done(us->srb);
884         us->srb = NULL;
885         scsi_unlock(host);
886     }
887     mutex_unlock(&us->dev_mutex);
888
889     /* Now we own no commands so it's safe to remove the SCSI host */
890     scsi_remove_host(host);

```

```

891 }
892
893 /* Second stage of disconnect processing: deallocate all resources */
894 static void release_everything(struct us_data *us)
895 {
896     usb_stor_release_resources(us);
897     dissociate_dev(us);
898
899     /* Drop our reference to the host; the SCSI core will free it
900      * (and "us" along with it) when the refcount becomes 0. */
901     scsi_host_put(us_to_host(us));
902 }

```

868 行，全文中仅有的两处设置 `US_FLIDX_DISCONNECTING` 这个 flag 的地方就在这里以及下面的这个 `usb_stor_release_resources()` 函数中。

870 行，`usb_stor_stop_transport(us)`，这个函数可是刚刚才讲过，就在 `command_abort()` 里调用。目的就是停掉当前的 `urb` 和 `sg`（如果有的话）。

871 行，`wake_up(&us->delay_wait)`，也已经讲过了，就是在 `device_reset()` 讲到的，当时在 `usb_stor_reset_common()` 中，会使用 `wait_event_interruptible_timeout()` 来进入睡眠，睡眠的目的是给 6 秒钟来让设备从 `reset` 状态恢复过来，但是如果在这期间我们要断开设备了，那么当然就没有必要再让那边继续睡眠了，设备都要断开了，还有什么恢复的意义呢？所以对于这种情况，我们回过头来看 `usb_stor_reset_common()`，会发现之后该函数立马从睡眠中醒来，清除掉为“reset”而设置的 flag，`US_FLIDX_RESETTING`，然后就返回了，返回值是 `FAILED`。

890 行，`scsi_remove_host()` 被调用，这是和最早的 `scsi_add_host` 相对应的，都是调用 SCSI Core 提供的函数。

再来看 `release_everything()`，896 行，`usb_stor_release_resources(us)`，这个则与我们当初 `usb_stor_acquire_resources(us)` 相对应。而 897 行的 `dissociate_dev(us)` 则和当初 `associate_dev()` 相对应。来看一下具体代码，来自 `drivers/usb/storage/usb.c`，把这两个函数的代码都一并“贴”出来：

```

816 /* Release all our dynamic resources */
817 static void usb_stor_release_resources(struct us_data *us)
818 {
819     US_DEBUGP("-- %s\n", __FUNCTION__);
820
821     /* Tell the control thread to exit. The SCSI host must
822      * already have been removed so it won't try to queue
823      * any more commands.
824      */
825     US_DEBUGP("-- sending exit command to thread\n");
826     set_bit(US_FLIDX_DISCONNECTING, &us->flags);
827     up(&us->sema);
828
829     /* Call the destructor routine, if it exists */
830     if (us->extra_destructor) {
831         US_DEBUGP("-- calling extra_destructor()\n");

```

```

832         us->extra_destructor(us->extra);
833     }
834
835     /* Free the extra data and the URB */
836     kfree(us->extra);
837     usb_free_urb(us->current_urb);
838 }
839
840 /* Dissociate from the USB device */
841 static void dissociate_dev(struct us_data *us)
842 {
843     US_DEBUGP("-- %s\n", __FUNCTION__);
844
845     kfree(us->sensebuf);
846
847     /* Free the device-related DMA-mapped buffers */
848     if (us->cr)
849         usb_buffer_free(us->push_dev, sizeof(*us->cr), us->cr,
850                         us->cr_dma);
851     if (us->iobuf)
852         usb_buffer_free(us->push_dev, US_IOBUF_SIZE, us->iobuf,
853                         us->iobuf_dma);
854
855     /* Remove our private data from the interface */
856     usb_set_intfdata(us->push_intf, NULL);
857 }

```

826 行，全文中仅有的两处设置 US\_FLIDX\_DISCONNECTING 这个 flag 的地方之一。

827 行，`up(&us->sema)`，知道这句是干什么的吗？还记得 `usb_stor_control_thread()`，当初讲到 `down_interruptible(&us->sema)`，就说该守护进程进入了睡眠，那么谁能把它唤醒，除了前面讲的 `queuecommand` 之外，这里同样也是唤醒它的代码。

继续往下，830 行，`struct us_data` 还有一个元素，`extra_data_destructor extra_destructor`，这是一个函数指针，实际上对某些设备来说，它们自己定义了一些额外的函数，在退出之前需要执行，比如 Datafab USB 紧凑读卡器。不过像这样的设备很少，对于大多数设备来说，这个函数指针都为空。如果定义了，那么就去执行它。

接下来 836 行，判断 `us->extra`，`us->extra` 就是为前面那个 `extra_data_destructor` 函数准备的参数，一般不会申请，如果申请过，那就释放它。

再接下来，`current_urb` 也没用了，释放吧。释放 `urb` 需要调用专门的函数 `usb_free_urb`，它来自 `drivers/usb/core/urb.c` 中。

到这里我们前面占有的资源基本上就释放掉了。

再看下一个函数，`dissociate_dev()`，很显然，这个函数和前面讲的 `associate_dev()` 函数是一对。在 `associate_dev` 函数中，我们调用了 `usb_buffer_alloc()` 函数，先后为 `us->cr` 和 `us->iobuf` 以及 `us->sensebuf` 分配了空间，所以这里首先调用相对应的函数 `usb_buffer_free()` 以及 `kfree()` 来释放这两段空间，然后，然后咱们曾经调用 `usb_set_intfdata` 来令 `us->push_intf` 的所对应的设备的

`driver_data` 指向了 `us`，所以这里 `usb_set_intfdata()` 再次调用从而让 `us->pusb_intf` 的 `driver_data` 指向空。

而 `release_everything()` 函数最后还有一个 `scsi_host_put()`，只是对这个 `host` 的引用计数减 1，如果该 `host` 的引用计数达到 0 了，那么将释放其对应的 `Scsi_Host` 数据结构所占的空间。一个 `scsi` 卡在其调用 `scsi_host_alloc` 时会被设置引用计数为 1，引用计数就是表示有多少进程使用了这个资源。

好了，终于，世界清静了。

最后，我们还有一点点内容要讲，那就是分析一下我们故事中的锁机制，至少我们曾经承诺过，要到最后才能讲锁机制，理由很简单，只有我们把整个故事都弄明白了，才可能弄清楚为什么在某个地方要加锁，因为锁机制永远都是牵连着重多代码的。

---

## 46. 其实世上本有路，走的人多了，也便没了路

我们前面说过，Linux 中，有信号量、自旋锁、互斥锁，自旋锁或者互斥锁从某种意义上来说就只是一种特殊的信号量，即信号量意味着资源数量有限，但这个有限也许可能有若干个，而锁反映的就是更加有限，限制到了数量为 1，即类似于所谓的“一夫一妻制”。它只属于你一个人，一旦你占有了它，如果别人还要想得到它，除非你释放。或者说除非你抛弃了它。

具体到 `usb-storage`，我们不管信号量和锁在 Linux 中是怎么实现的，它们之间是否有区别对我们来说也无所谓，事实上，`usb-storage` 中使用的都是锁，即便早期的 Linux 内核中是信号量，也是把它当成锁来使用，那时候的内核中都是把信号量的初始化为 1。当信号量初始化为 1，那么对我们来说，它就相当于退化为一把锁了。而锁只有两种状态，上锁和解锁。

此前我们见过很多次两组函数，但一直绝口不提。

它们就是 `scsi_lock()` 和 `scsi_unlock()`，以及 `mutex_lock(&(us-> dev_mutex))` 和 `mutex_unlock(&(us-> dev_mutex))`。

先来看第一组。它们是我们自己定义的宏。来自 `drivers/usb/storage/usb.h`：

```
173 /* The scsi_lock() and scsi_unlock() macros protect the sm_state and the
174  * single queue element srb for write access */
175 #define scsi_unlock(host)      spin_unlock_irq(host->host_lock)
176 #define scsi_lock(host)       spin_lock_irq(host->host_lock)
```

显然，这两个函数是一对。而它们的作用，就是利用自旋锁来保护资源，这把锁就是 `struct Scsi_Host` 的一个成员 `spinlock_t *host_lock`。那么在什么情况下需要使用这两个自旋锁函数来保护资源呢？

当你要写 `us->srb` 时，（不是写 `us->srb` 的元素，而是写 `us->srb`，比如令 `us->srb=NULL`），你需要使用这把自旋锁。另一种情况是，当你调用 `scsi mid layer`（`scsi` 中层）的函数时，有时候这些函数要求调用者拥有这把锁，即 `host_lock`。

搜索一下，发现一共有 8 处使用了 `scsi_lock`。

我们列举一些。

第一处，`drivers/usb/storage/usb.c:usb_stor_release_resources()`函数中：

```
880     if (us->srb) {
881         us->srb->result = DID_NO_CONNECT << 16;
882         scsi_lock(host);
883         us->srb->scsi_done(us->srb);
884         us->srb = NULL;
885         scsi_unlock(host);
886     }
```

这个不用解释了吧，赤裸裸地写 `us->srb`，自然要用 `scsi_lock/scsi_unlock`。

第二处，`drivers/usb/storage/usb.c: storage_post_reset ()`函数中：

```
247     /* Report the reset to the SCSI core */
248     scsi_lock(us_to_host(us));
249     usb_stor_report_bus_reset(us);
250     scsi_unlock(us_to_host(us));
```

这就是第二种情况，因为这里调用了 `usb_stor_report_bus_reset()`，这个函数来自 `drivers/usb/storage/scsiglue.c`，也是我们定义的：

```
325 void usb_stor_report_bus_reset(struct us_data *us)
326 {
327     scsi_report_bus_reset(us_to_host(us), 0);
328 }
```

注意到，它里面调用了 `scsi_report_bus_reset()`，后者来自 `drivers/scsi/scsi_error.c`，这正是 `SCSI mid layer` 定义的函数，这个函数的注释说得很清楚，调用它时必须要有 `host lock`。

有人说还有第三种情况，`drivers/usb/storage/usb.c:usb_stor_control_thread()`函数中：

```
323     /* lock access to the state */
324     scsi_lock(host);
325
326     /* has the command timed out *already* ? */
327     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
328         us->srb->result = DID_ABORT << 16;
329         goto SkipForAbort;
330     }
331
332     scsi_unlock(host);
```

这段代码无非是调用了 `test_bit`，但是却使用了 `scsi_lock/scsi_unlock` 这对冤家，这是什么

原因？其实是这样的，我们注意到这里有一个 goto 语句：

```

390 SkipForAbort:
391     US_DEBUGP("scsi command aborted\n");
392 }
393
394     /* If an abort request was received we need to signal that
395     * the abort has finished. The proper test for this is
396     * the TIMED_OUT flag, not srb->result == DID_ABORT, because
397     * the timeout might have occurred after the command had
398     * already completed with a different result code. */
399     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
400         complete(&(us->notify));
401
402         /* Allow USB transfers to resume */
403         clear_bit(US_FLIDX_ABORTING, &us->flags);
404         clear_bit(US_FLIDX_TIMED_OUT, &us->flags);
405     }
406
407     /* finished working on this command */
408     us->srb = NULL;
409     scsi_unlock(host);

```

看 408 行，`us->srb=NULL`，还是老一套。这才是为什么之前要用 `scsi_lock` 的真正原因。因为我要跳转，而跳过去以后需要写 `us->srb`，所以要获得 `host lock`，当然，这一段测试超时的代码本身被使用上的可能性也不大，因为这里命令还没有开始执行呢，试想，一个命令还没有开始就超时。

再来看第二组，关于 `us-> dev_mutex`，它体现出来的是一种进程间的同步机制。2.6.22 的内核代码中一共有 9 处使用了这把锁。这把锁的存在基本上保证了 `us` 被合理使用，也就是说但凡要修改 `us` 的成员时，都要拿到这把锁，改完之后就要释放锁。

一种最典型的情景就是，当 `usb_stor_control_thread()` 正在执行命令，那么当然 `quiesce_and_remove_host()` 函数就不能释放资源了，得等你当前这个命令执行完了，才会去执行清除 `srb` 的代码。就好比你在考场上战战兢兢地答题，老师却强行把你的试卷收上去，你说你会不会很愤怒？

总之，正是因为有了信号量和自选锁这样的东西，才保证了整个操作系统正常运转，如果谁违规了，那么伤害的是大家的利益。这也就是 Linux 内核的同步机制。

好了。我的故事讲完了。蓦然回首，发现，其实，我一直在寻觅，寻觅这个故事的结局，寻觅自己灵魂的出路，最终，追寻到了前者，却一直没有找到后者。

# 附 录

## Linux 那些事儿之我是 sysfs

1. sysfs 初探 .....	526	3.1.3 super_block 与 vfsmount .....	552
2. 设备模型 .....	527	3.2 sysfs .....	553
2.1 设备底层模型 .....	528	3.2.1 sysfs_dirent .....	553
2.1.1 kobject .....	528	3.2.2 sysfs_create_dir() .....	554
2.1.2 kset .....	530	3.2.3 sysfs_create_file() .....	556
2.1.3 kobj_type .....	531	3.3 file_operations .....	557
2.2 设备模型上层容器 .....	532	3.3.1 示例一：读入 sysfs 目录 的内容 .....	558
2.3 示例一：usb 子系统 .....	535	3.3.2 示例二：读入 sysfs 普通 文件的内容 .....	560
2.4 示例二：usb storage 驱动 .....	540		
3. sysfs 文件系统 .....	546		
3.1 文件系统 .....	547		
3.1.1 dentry & inode .....	548		
3.1.2 一起散散步 path_walk .....	551		



## 1. sysfs 初探

内核文档 `Documentation/filesystems/sysfs.txt` 中写到,“sysfs 是一个虚拟文件系统(类似 proc 文件系统),用于将系统中的设备组成层次结构,并向用户模式程序提供详细的内核数据结构信息。”

到目录 `/sys` 看一看:

```
localhost:/sys#ls /sys/  
block/ bus/ class/ devices/ firmware/ kernel/ module/ power/
```

- `block` 目录: 包含所有的块设备。
- `devices` 目录: 包含系统所有的设备,并根据设备挂接的总线类型组织成层次结构。
- `bus` 目录: 包含系统中所有的总线类型。
- `drivers` 目录: 包括内核中所有已注册的设备驱动程序。
- `class` 目录: 系统中的设备类型(如网卡设备,声卡设备等)。

`/sys` 下面的目录和文件反映了整台机器的系统状况。比如 `bus` 目录:

```
localhost:/sys/bus#ls  
i2c/ ide/ pci/ pci_express/ platform/ pnp/ scsi/ serio/ usb/
```

这里面包含了系统用到的一系列总线,包括 `pci`, `ide`, `scsi`, `usb` 等,比如可以在 `usb/` 文件夹中发现 U 盘、USB 鼠标的信息。

`sysfs` 是一个虚拟的文件系统。那么什么叫文件系统? 文件系统是一个很模糊又很广泛的概念,狭义理解“文件”是指磁盘文件,广义理解,可以是有组织有次序地存储与任何介质(包括内存)的一组信息。

Linux 把所有的资源都看成文件,让用户通过一个统一的文件系统操作接口,也就是同一组系统调用,对属于不同文件系统的文件进行操作。这样,就可以对用户程序隐藏各种不同文件系统的实现细节,为用户程序提供了一个统一、抽象、虚拟的文件系统接口,这就是所谓“VFS (Virtual Filesystem Switch)”。这个抽象出来的接口就是一组函数操作。

我们要讨论一个文件系统,首先要知道这个文件系统的信息来源在哪里。所谓信息来源是指文件组织存放的地点。比如,使用 `mount` 命令挂载一个分区时:

```
mount -t vfat /dev/hda2 /mnt/c
```

我们就知道挂载在/mnt/c 下的是一个 vfat 类型的文件系统，它的信息来源是在第一块硬盘的第 2 个分区。但是，sysfs 的挂载过程这样被挂载的：

```
mount -t sysfs sysfs /sys
```

好像看不出它的信息来源在哪。因为 sysfs 是一个虚拟文件系统，并没有一个实际存放文件的介质，断电后就完了。简而言之，sysfs 的信息来源是设备的层次结构，读一个 sysfs 文件，就是动态地从设备树寻找到相关节点，提取信息，返回给用户。

所以，首先我要先讲一讲 sysfs 文件系统的信息来源——设备层次结构，即设备是如何被组织关联起来的，就是 Linux 的设备模型。因此，我们先学习 Linux 的设备模型，然后学习文件系统，特别是 sysfs 文件系统。

## 2. 设备模型

何谓设备模型？我们首先要知道这个设备模型是用来干什么的，为什么要为设备建立一个模型。建立模型的意义就在于方便管理。最初 Linux 建立设备模型的初衷很直白——为了省电。建立一个全局的设备树（device tree），当系统进入休眠时，系统可以通过这颗树找到所有的设备，随时让它们挂起（suspend）或是唤醒（resume）。说简单点儿，就是实现一些动态电源管理的功能。

当热插拔（hot plug）技术出现后，设备模型还要能够动态地反映出设备的变化情况，不能只在开机时遍历（populate）一遍就了事。而且就现在而言，一个电脑上的设备种类五花八门，硬盘、光驱、鼠标、U 盘、键盘等。同一种设备也可以很多个，比如可以同时使用几个 U 盘。要高效率地管理这么多设备，设计出一个良好、清晰的设备模型尤为重要。一个好的设计（design）在于事物都分门别类，井井有条。2.6 内核中的设备模型就是一个很好的设计，它支持如下功能：

（1）电源管理和系统关机。

（2）与用户空间通信。这些通信的操作都是通过 sysfs 文件系统这个中介来完成的。sysfs 文件系统紧密团结在设备模型周围，把控制设备的接口（比如设备休眠，唤醒）通过文件的形式暴露给人民群众（用户空间）。可以通过读写 sysfs 中文件的方式，读取或改变设备的配置。

（3）支持热插拔设备（这个必须有）。

（4）设备分类。把设备分门别类有助于设备的管理和使用。比如要找 USB 鼠标，只要去 classes/input/里面去找就可以了，而不必去关心这个鼠标是接到哪个 USB 主控制器的哪个 Hub

的第几个端口上的。

(5) 对象生命周期。上面描述的功能，包括支持热插拔和 `sysfs`，使得这个模型相当复杂。得有一个好的机制来实现设备生命周期的管理。比如说，把 USB 鼠标拔了之后，全局设备树和 `sysfs` 里面得相应去掉。

好了，总结一下，Linux 设备模型是相当复杂的，任务是艰巨的，但好在人民群众的力量是无穷的。我们此时也采用自底而上的方法，来学习 Linux 设备模型。

## 2.1 设备底层模型

上层可以指挥控制底层，底层就是螺丝钉、板砖块，安安心心地等待任务，接受上层的命令。

### 2.1.1 kobject

Linux 设备模型的底层是数据结构 `kobject`，在内核文档中有一个专门的文件 `Documentation/kobject.txt` 来介绍。

上面说过，设备模型的首要任务在于把设备连在一起，形成一个清晰明朗的树状结构。对于设备而言，虽然设备的种类千奇百怪，但是有一些属性是都要有的，比如引用计数、设备上锁、设备名称，用来形成树状结构的链表指针等。于是人们把这些公共的东西都放在一个数据结构中，便于管理，这个数据结构就叫做 `kobject`。

`kobject`，顾名思义，kernel 对象，这是用 C 语言实现了面向对象的套路（继承）。`kobject` 就类似抽象的基类。它只被嵌入更大的对象（Linux 里叫 `container`，容器）中，如 `bus`，`devices`，`drivers` 等数据结构。而且给定 `kobject`，我们可以通过一个叫 `container_of` 的宏，来反查它所在的数据结构。

```
container_of(pointer, type, member)
```

实际上 `container_of` 做了一件简单但却非常实用有效的东西。比如给定一个结构 `A_t`：

```
struct A_t{
    char a;
    int b;
};
struct A_t A;
int * b_p = &(A.b);
```

使用如下方法就可以从数据结构的一个元素出发，得到整个数据结构的指针。

```
struct A_t * A_p = container_of(b_p, struct A_t, b);
```

`container_of` 这个宏在 `kernel.h` 文件中定义，具体的实现也不是很复杂。有了它省了不少事，不然我们得为每一个元素设置一个指向父数据结构的指针。Linux 开创了 `container_of` 技术，带

领人们走向了新时代。

kobject 结构定义为：

```
struct kobject {
    char * k_name; //指向设备名称的指针
    char name[KOBJ_NAME_LEN]; //设备名称
    struct kref kref; //对象引用计数
    struct list_head entry; //挂接到所在 kset 中去的单元
    struct kobject * parent; //指向父对象的指针
    struct kset * kset; //所属 kset 的指针
    struct kobj_type * ktype; //指向其对象类型描述符的指针
    struct dentry * dentry; //sysfs 文件系统中与该对象对应的文件节点路径指针
    wait_queue_head_t poll;
};
```

首先是关于 `k_name` 和 `name`，如果名称比较短，就直接存到 `char name[KOBJ_NAME_LEN]` 中，`k_name` 就直接指向 `name`。如果名称很长，`name` 存不下，那么就再重新申请一块地方，让 `k_name` 指向那里。所以 `k_name` 一直都是会指向设备名称的。要是没有 `name` 数组，就得每次都得动态申请。但光有 `name` 也不行，因为我们没法估计设备名称最长有多长，总有那么些 BT 把名字取得巨长无比。

`kref` 没什么好讲的，就是计数。告诉别人这个东西有几个人在用，闲人莫扰。内核提供两个函数 `kobject_get()`，`kobject_put()` 分别用于增加和减少引用计数。`entry`，`parent`，`kset` 就是用来形成树状结构指针。`kobj_type * ktype` 用来表明该 `kobject` 的类型。而 `dentry` 就是和 `sysfs` 联系紧密的文件节点路径。每个在内核中注册的 `kobject` 对象都对应于 `sysfs` 文件系统中的—个目录，`dentry` 就是连接两个系统的桥梁。这些我们会在下面的内容—讲到。

kobject 接口函数：

```
void kobject_init(struct kobject * kobj);
```

`kobject` 初始化函数。设置 `kobject` 引用计数为 1，`entry` 指向自身，其所属 `kset` 引用计数加 1。

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

设置指定 `kobject` 的名称。

```
void kobject_cleanup (struct kobject * kobj), void kobject_release (struct kref *kref);
```

`kobject` 清除函数。当其引用计数为 0 时，释放对象占用的资源。

```
struct kobject *kobject_get(struct kobject *kobj);
```

将 `kobj` 对象的引用计数加 1，同时返回该对象的指针。

```
void kobject_put(struct kobject * kobj);
```

将 `kobj` 对象的引用计数减 1，如果引用计数降为 0，则调用 `kobject_release()` 释放该 `kobject` 对象。

```
int kobject_add(struct kobject * kobj);
```

将 `kobj` 对象加入 Linux 设备层次。挂接该 `kobject` 对象到 `kset` 的 `list` 链中，增加父目录各级 `kobject` 的引用计数，在其 `parent` 指向的目录下创建文件节点，并启动该类型内核对象的 `hotplug` 函数。

```
int kobject_register(struct kobject * kobj);
```

`kobject` 注册函数。通过调用 `kobject_init()` 初始化 `kobj`，再调用 `kobject_add()` 完成该内核对象的注册。

```
void kobject_del(struct kobject * kobj);
```

从 Linux 设备层次(hierarchy)中删除 `kobj` 对象。

```
void kobject_unregister(struct kobject * kobj);
```

`kobject` 注销函数。与 `kobject_register()` 相反，它首先调用 `kobject_del` 从设备层次中删除该对象，再调用 `kobject_put()` 减少该对象的引用计数，如果引用计数降为 0，则释放该 `kobject` 对象。

## 2.1.2 kset

`kset`，顾名思义，是 `kobject` 的集合。`kobject` 通过 `kset` 组织成层次化的结构，`kset` 是具有相同类型的 `kobject` 的集合。`kobject` 相当于叶子节点，`kset` 类似于内节点，二者连接形成了一个树状结构。`kset` 数据结构如下：

```
struct kset {
    struct kobj_type * ktype; //指向该 kset 对象类型描述符的指针
    struct list_head list; //用于连接该 kset 中所有 kobject 的链表头
    spinlock_t list_lock; //用于互斥访问
    struct kobject kobj; //嵌入的 kobject
    struct kset_uevent_ops * uevent_ops;
};
```

包含在 `kset` 中的所有 `kobject` 被组织成一个双向循环链表，`list` 正是该链表的头。`ktype` 指向一个 `kobj_type` 结构，表示这些对象的类型，该 `kset` 中的所有 `kobject` 共享之。我们注意到 `kobject` 结构中也有 `kobj_type`，但 `kset` 中的 `ktype` 优先级较高，`kobject` 会利用自己的 `*kset` 找到自己所属的 `kset`，并把 `*ktype` 指定成该 `kset` 下的 `ktype`，除非没有定义 `kset`，`kobject` 才会用自己的 `ktype` 来建立关联。`kset` 数据结构还内嵌了一个 `kobject` 对象(由 `kobj` 表示)，所有属于这个 `kset` 的 `kobject` 对象的 `parent` 域均指向这个内嵌的对象。此外，`kset` 还依赖于 `kobj` 维护引用计数：`kset` 的引用计数实际上就是内嵌的 `kobject` 对象的引用计数。如图 1 所示反映了 `kset` 和 `kobject` 的关系。

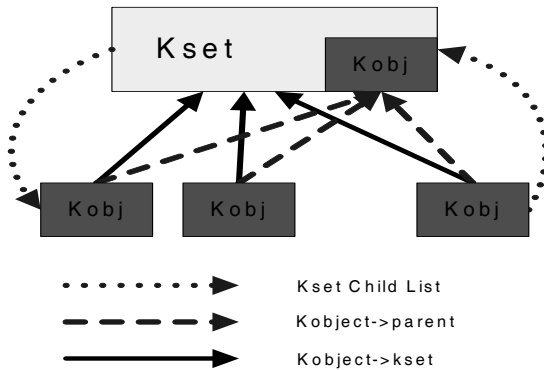


图 1 kset 和 kobject 关系

有相同类型（`kobj_type`）的 `kobject` 集合在一起组成 `kset`，许多 `kset`（可以是不同类型）集合在一起组成了子系统（`subsystem`）。在那遥远的年代，有一个专门的数据结构 `subsystem` 描述子系统，在改革开放的新时代，这个数据结构已经消失了。因为有 `kset`, `kobject` 足以完全描述整个设备树。

在 `kobject.h` 里的有个宏，来声明子系统。实际上就是声明一个 `kset`。

```

#define decl_subsys(_name, _type, _uevent_ops) \
struct kset _name##_subsys = { \
    .kobj = { .name = __stringify(_name) }, \
    .ktype = _type, \
    .uevent_ops = _uevent_ops, \
}
  
```

`kset` 接口函数与 `kobject` 相似，`kset_init()` 完成指定 `kset` 的初始化，`kset_get()` 和 `kset_put()` 分别增加和减少 `kset` 对象的引用计数。`kset_add()` 和 `kset_del()` 函数分别实现将指定 `kset` 对象加入设备层次和从其中删除；`kset_register()` 函数完成 `kset` 的注册而 `kset_unregister()` 函数则完成 `kset` 的注销。

### 2.1.3 kobj\_type

`kobject` 是嵌入到 `device`, `driver`, `bus` 这些大容器中的数据结构，每个容器也会有很多不同的属性可供配置。`kobj_type` 正是说明 `kobject` 类型的结构。`kobj_type` 数据结构如下：

```

struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops * sysfs_ops;
    struct attribute ** default_attrs;
};
  
```

其中，`release` 是注销函数指针。`attribute` 是该 `kobject` 的默认属性列表。在 `sysfs` 中，`kobject` 对应目录，属性对应文件。属性数据结构如下：

```

struct attribute {
    const char      * name;
    struct module   * owner;
    mode_t          mode;
};

```

attribute 以文件的形式输出到 sysfs 的目录当中，文件名就是 name。kobject 是连接上述容器于 sysfs 之间的桥料，由于属性不同，kobj\_type 与 sysfs 的接口函数也会有所不同。文件读写的方法对应于 kobj\_type 中的 sysfs\_ops。读写 sysfs 一个目录下的文件，就是读写该设备的属性。

sysfs\_ops 是指向如何读写的函数指针。sysfs 操作表包括两个函数 store() 和 show()，对应文件的读/写。当用户态读取属性时，show() 函数被调用，该函数把指定属性值存入 buffer 中返回给用户空间；而 store() 函数用于存储用户空间传入的属性值。

## 2.2 设备模型上层容器

上面说的是 Linux 中的板砖，现在来学习使用这些板砖的高级货：bus, device, device\_driver。Linux 设备模型通过总线（bus），设备（device），驱动（device\_driver）这三个数据结构来描述的。总线是处理器和一个或多个设备之间的通道。在设备模型中，所有设备都通过总线连接。即使有些设备并没有连接到一根物理上的总线上，我们也会为其设置一个内部的、虚拟的“平台”总线，来维持总线、设备、驱动的三角关系。总线可以插入另一个总线，比如一个 USB 控制器常常是一个 PCI 设备。总而言之，设备模型表示了设备、驱动和总线之间的连接关系。

我们说一个设备（或驱动），实际上要说明的是属于某个总线系统下的设备（或驱动）。比如有单说网卡驱动，这个就没表达准确，还要说明在哪个总线下的网卡驱动，是 USB 总线下的网卡驱动，还是 PCI 总线下的网卡驱动。

设备和驱动总在互相寻找自己的另一半，首要原则是门当户对，即大家都得属于同一个 bus\_type。一个 PCI 网卡的驱动是不可能驱动一个 U 盘。另外设备和驱动得匹配，其实设备和驱动从一出生就写好了今生只能和什么的人在一起。设备老早就把自身条件和择偶要求烙在了心坎里，它要做的就是耐心的等待和寻找。一旦找了匹配的另一半，二者就会绑定在一起，永不分离。然而，驱动是一个花心大萝卜，因为一个驱动程序可以支持很多个设备。

总线使用结构 struct bus\_type 描述，定义如下：

```

struct bus_type {
    const char      * name; //总线类型的名称
    struct module   * owner;

    struct kset      subsys; //该总线的 subsystem
    struct kset      drivers; //所有与该总线相关的驱动程序集合
    struct kset      devices; //所有挂接在该总线上的设备集合
    struct klist     klist_devices;
    struct klist     klist_drivers;
};

```

```

struct blocking_notifier_head bus_notifier;

struct bus_attribute * bus_attrs; //总线属性
struct device_attribute * dev_attrs; //设备属性
struct driver_attribute * drv_attrs; //驱动属性
struct bus_attribute drivers_autoprobe_attr;
struct bus_attribute drivers_probe_attr;

int      (*match)(struct device * dev, struct device_driver * drv);
int      (*uevent)(struct device *dev, char **envp,
                  int num_envp, char *buffer, int buffer_size);
int      (*probe)(struct device * dev);
int      (*remove)(struct device * dev);
void     (*shutdown)(struct device * dev);

int      (*suspend)(struct device * dev, pm_message_t state);
int      (*suspend_late)(struct device * dev, pm_message_t state);
int      (*resume_early)(struct device * dev);
int      (*resume)(struct device * dev);
unsigned int drivers_autoprobe:1;
};

```

仔细查看这个结构，`subsys` 描述该总线的子系统，`subsys` 是一个 `kset` 结构，它连接到一个全局变量 `kset bus_subsys` 中。这样，每一根总线系统都会通过 `bus_subsys` 结构连接起来。`kset device` 是该总线系统里所有设备的集合，`kset driver` 是该总线系统里所有驱动的集合。该总线里的设备用 `klist` 指针连成了一个链表，`klist_devices` 则指向这个设备链表。该总线里的驱动也用 `klist` 指针连成了一个链表，`klist_drivers` 则指向这个驱动链表。毕竟每次都用 `kset device`，`kset driver` 来遍历所有设备/驱动很麻烦，还是用 `klist` 指针比较直接。`bus_type` 结构还包含了一些函数（`match()`、`hotplug()`等），处理相应的热插拔、即插即拔和电源管理事件。

在 `sysfs` 文件系统中，我们可以清晰地看到它们之间的联系。`kset bus_subsys` 对应于 `/sys/bus/` 这个目录。每个 `bus_type` 对象都对应 `/sys/bus` 目录下的一个子目录，如 `PCI` 总线类型对应于 `/sys/bus/pci`。在每个这样的目录下都存在两个子目录：`devices` 和 `drivers`（分别对应于 `bus_type` 结构中的 `devices` 和 `drivers` 元素）。其中 `devices` 子目录描述连接在该总线上的所有设备，而 `drivers` 目录则描述与该总线关联的所有驱动程序。

设备使用结构 `struct device` 描述，定义如下：

```

struct device {
    struct klist      klist_children;
    struct klist_node knode_parent;      /* node in sibling list */
    struct klist_node knode_driver;
    struct klist_node knode_bus;
    struct device     *parent;

    struct kobject kobj;
    char bus_id[BUS_ID_SIZE]; /* position on parent bus */
    struct device_type *type;
    unsigned is_registered:1;
    unsigned uevent_suppress:1;
    struct device_attribute uevent_attr;
};

```



```

struct device_attribute *devt_attr;

struct semaphore sem; /* semaphore to synchronize calls to
                       * its driver. */

struct bus_type * bus; /* type of bus device is on */
struct device_driver *driver; /* which driver has allocated this device */
void *driver_data; /* data private to the driver */
void *platform_data;
struct dev_pm_info power;

#ifdef CONFIG_NUMA
int numa_node; /* NUMA node this device is close to */
#endif
u64 *dma_mask; /* dma mask (if dma'able device) */
u64 coherent_dma_mask; /* Like dma_mask, but for
                        alloc_coherent mappings as
                        not all hardware supports
                        64 bit addresses for consistent
                        allocations such descriptors. */

struct list_head dma_pools; /* dma pools (if dma'ble) */

struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                                override */
/* arch specific additions */
struct dev_archdata archdata;

spinlock_t devres_lock;
struct list_head devres_head;

/* class_device migration path */
struct list_head node;
struct class *class;
dev_t devt; /* dev_t, creates the sysfs "dev" */
struct attribute_group **groups; /* optional groups */
void (*release)(struct device * dev);
};

```

我们首先要关心的是 `struct bus_type * bus` 和 `struct device_driver *driver` 这两个元素，它们分别对应于设备所属的总线和驱动。`kobject kobj` 用于引用计数管理并通过它实现设备层次结构。`klist_children` 指向该 `device` 对象子对象链表头，`parent` 则指向父对象。`knode_parent`, `knode_driver`, `knode_bus` 分别是挂入 `parent`, 驱动, 总线链表中的指针。这就像社会中的人一样，首先我是一个中国人，我有身份证号码，通过身份证号码可以找到我。此外我还曾是复旦大学的学生，我还有学生证，学生证上有的号码，通过这个号码就能在复旦大学里找到我。这些号码就跟这些指针一样。一个人一出生就不再是孤立的个体，它已经被各种关系连入到整个社会的网络中，如父母、国籍、学校。设备也是一样，设备从一开始就得向各个组织机构（如总线，驱动）报到。

内核提供了相应的函数用于操作 `device` 对象。其中 `device_register()` 函数将一个新的 `device` 对象插入设备模型，并自动在 `/sys/devices` 下创建一个对应的目录。`device_unregister()` 完成相反的操作，注销设备对象。`get_device()` 和 `put_device()` 分别增加与减少设备对象的引用计数。通常

device 结构不单独使用，而是包含在更大的结构中作为一个子结构使用，比如描述 USB 设备的 struct usb\_device。

驱动使用结构 struct device\_driver 描述，定义如下：

```
struct device_driver {
    const char      * name; //设备驱动程序的名称
    struct bus_type  * bus; //该驱动所管理的设备挂接的总线类型

    struct kobject    kobj; //内嵌 kobject 对象
    struct klist      klist_devices; //该驱动所管理的设备链表头
    struct klist_node  knode_bus; //挂入总线链表中的指针

    struct module     * owner;
    const char      * mod_name; /* used for built-in modules */
    struct module_kobject * mkobj;

    int (*probe)(struct device * dev);
    int (*remove) (struct device * dev);
    void (*shutdown) (struct device * dev);
    int (*suspend) (struct device * dev, pm_message_t state);
    int (*resume) (struct device * dev);
};
```

与 device 结构类似，device\_driver 对象依靠内嵌的 kobject 对象实现引用计数管理和层次结构组织。klist\_devices，knode\_bus 是链入整个层次结构的指针。

## 2.3 示例一：usb 子系统

在 usb 子系统的初始化函数 usb\_init 里，调用了 bus\_register() 函数：

```
static int __init usb_init(void)
{
    □□
    retval = bus_register(&usb_bus_type);
    □□
}
```

在 USB Core 部分中，没有深入这个函数。现在来仔细观察一下 bus\_register(&usb\_bus\_type)。这个函数的作用是向系统注册 usb\_bus\_type 这个总线类型。usb\_bus\_type 在 usb core 中详细阐述过，此不赘述。bus\_register 代码如下：

```
807 int bus_register(struct bus_type * bus)
808 {
809     int retval;
810
811     BLOCKING_INIT_NOTIFIER_HEAD(&bus->bus_notifier);
812
813     retval = kobject_set_name(&bus->subsys.kobj, "%s", bus->name);
814     if (retval)
815         goto out;
816
```

```

817     subsys_set_kset(bus, bus_subsys);
818     retval = subsystem_register(&bus->subsys);
819     if (retval)
820         goto out;
821
822     kobject_set_name(&bus->devices.kobj, "devices");
823     bus->devices.kobj.parent = &bus->subsys.kobj;
824     retval = kset_register(&bus->devices);
825     if (retval)
826         goto bus_devices_fail;
827
828     kobject_set_name(&bus->drivers.kobj, "drivers");
829     bus->drivers.kobj.parent = &bus->subsys.kobj;
830     bus->drivers.ktype = &ktype_driver;
831     retval = kset_register(&bus->drivers);
832     if (retval)
833         goto bus_drivers_fail;
834
835     klist_init(&bus->klist_devices, klist_devices_get,
836               klist_devices_put);
837     klist_init(&bus->klist_drivers, NULL, NULL);
838
839     bus->drivers_autoprobe = 1;
840     retval = add_probe_files(bus);
841     if (retval)
842         goto bus_probe_files_fail;
843
844     retval = bus_add_attrs(bus);
845     if (retval)
846         goto bus_attrs_fail;
847
848     pr_debug("bus type '%s' registered\n", bus->name);
849     return 0;
850 bus_attrs_fail:
851     remove_probe_files(bus);
852 bus_probe_files_fail:
853     kset_unregister(&bus->drivers);
854 bus_drivers_fail:
855     kset_unregister(&bus->devices);
856 bus_devices_fail:
857     subsystem_unregister(&bus->subsys);
858 out:
859     return retval;
860 }

```

811 行，是关于内核异步通信机制。

813 行，把 `usb_bus_type` 子系统的名字设为 `bus->name ("usb")`，就是调用 `vsnprintf()`。

817 行，`subsys_set_kset(bus, bus_subsys)`，这是一个宏，

```

#define subsys_set_kset(obj, _subsys) \
    (obj)->subsys.kobj.kset = &(_subsys)

```

把 `usb_bus_type->subsys.kobj.kset` 指向全局变量 `kset bus_subsys`，`bus_subsys` 这个全局变量是在 `usb.c` 文件中定义。

```
146 static decl_subsys(bus, &ktype_bus, NULL);
```

decl\_subsys 宏的定义在 kobject.h 文件:

```
#define decl_subsys(_name, _type, _uevent_ops) \
struct kset _name##_subsys = { \
    .kobj = { .name = __stringify(_name) }, \
    .ktype = _type, \
    .uevent_ops = _uevent_ops, \
}
```

在 817 行设置了 `usb_bus_type->subsys.kobj.kset = bus_subsys.kset` 之后, 马上 818 行, `subsystem_register(&bus->subsys)`, 向全局的 `bus_subsys` “登记”, 把自己加入到 `bus_subsys` 的链表中去。深入 `subsystem_register` 函数查看, 函数的 trace 如下, 最终调用了 `kobject_shadow_add()` 函数,

```
subsystem_register() -> kset_register -> kset_add() ->
kobject_add()->kobject_shadow_add()
```

`kobject_shadow_add()` 函数如下, 作用是把一个 `kobject` 加入到层次结构中去。

```
165 int kobject_shadow_add(struct kobject * kobj, struct dentry *shadow_parent)
166 {
167     int error = 0;
168     struct kobject * parent;
169
170     if (!(kobj = kobject_get(kobj)))
171         return -ENOENT;
172     if (!kobj->k_name)
173         kobj->k_name = kobj->name;
174     if (!*kobj->k_name) {
175         pr_debug("kobject attempted to be registered with no name!\n");
176         WARN_ON(1);
177         kobject_put(kobj);
178         return -EINVAL;
179     }
180     parent = kobject_get(kobj->parent);
181
182     pr_debug("kobject %s: registering. parent: %s, set: %s\n",
183             kobject_name(kobj), parent ? kobject_name(parent) : "<NULL>",
184             kobj->kset ? kobj->kset->kobj.name : "<NULL>" );
185
186     if (kobj->kset) {
187         spin_lock(&kobj->kset->list_lock);
188
189         if (!parent)
190             parent = kobject_get(&kobj->kset->kobj);
191
192         list_add_tail(&kobj->entry, &kobj->kset->list);
193         spin_unlock(&kobj->kset->list_lock);
194         kobj->parent = parent;
195     }
196
197     error = create_dir(kobj, shadow_parent);
198     if (error) {
199         /* unlink does the kobject_put() for us */
```

```

200         unlink(kobj);
201         kobject_put(parent);
202
203         /* be noisy on error issues */
204         if (error == -EEXIST)
205             printk(KERN_ERR "kobject_add failed for %s with "
206                        "-EEXIST, don't try to register things with "
207                        "the same name in the same directory.\n",
208                        kobject_name(kobj));
209         else
210             printk(KERN_ERR "kobject_add failed for %s (%d)\n",
211                    kobject_name(kobj), error);
212         dump_stack();
213     }
214
215     return error;
216 }

```

186 行到 195 行就是把自己连入到父辈的上级 `kset` 中。我们注意到在 197 行调用了 `create_dir(kobj)`，这个函数的作用是在 `sysfs` 下创建一个文件夹，可见 `kobject` 和 `sysfs` 是同时更新的。

弄明白了 `subsystem_register()`，继续回到 `bus_register()` 往下走。822 行至 833 行的代码还是一样的套路，把 `bus->devices` 和 `bus->drivers` 这两个 `kset` 链入 `bus->subsys` 中去。835 行，836 行初始化 `bus->klist_devices` 和 `bus->klist_drivers` 两个链表。`klist_devices/klist_drivers`，顾名思义，所有属于这条总线的设备/驱动都会出现这链表上。835 行 `klist_devices_get`，`klist_devices_put` 是操作设备引用计数器的 `get` 和 `put` 函数。

838 行，将 `bus->drivers_autoprobe` 设置为 1。`drivers_autoprobe` 是干什么的呢？它其实是个标志位，如果为 1，那么意味着，每次我们向该总线注册一个设备时，会自动去 `bus->klist_drivers` 里面去寻找 `match` 自己的另一半。每次我们向该总线注册一个驱动时，也会自动去 `bus->klist_devices` 里面去寻找与 `match` 自己的另一半。当 `drivers_autoprobe = 0` 时，我们不做自动探测，每次加了一个设备或是驱动，不去找适合自己的另一半。

839 行，通过 `sysfs` 把 `drivers_autoprobe` 这个变量暴露给用户。`add_probe_files` 函数定义如下：

```

559 static int add_probe_files(struct bus_type *bus)
560 {
561     int retval;
562
563     bus->drivers_probe_attr.attr.name = "drivers_probe";
564     bus->drivers_probe_attr.attr.mode = S_IWUSR;
565     bus->drivers_probe_attr.attr.owner = bus->owner;
566     bus->drivers_probe_attr.store = store_drivers_probe;
567     retval = bus_create_file(bus, &bus->drivers_probe_attr);
568     if (retval)
569         goto out;
570
571     bus->drivers_autoprobe_attr.attr.name = "drivers_autoprobe";

```

```

572     bus->drivers_autoprobe_attr.attr.mode = S_IWUSR | S_IRUGO;
573     bus->drivers_autoprobe_attr.attr.owner = bus->owner;
574     bus->drivers_autoprobe_attr.show = show_drivers_autoprobe;
575     bus->drivers_autoprobe_attr.store = store_drivers_autoprobe;
576     retval = bus_create_file(bus, &bus->drivers_autoprobe_attr);
577     if (retval)
578         bus_remove_file(bus, &bus->drivers_probe_attr);
579 out:
580     return retval;
581 }

```

我们前面说过，sysfs 里面，kobject 对应目录，attribute 对应文件。563 行到 569 行就是创建了一个名为 `drivers_probe` 的文件，这个文件不可读，只可写。文件的写操作，对应与函数 `store_drivers_probe`。

```

230 static ssize_t store_drivers_probe(struct bus_type *bus,
231                                   const char *buf, size_t count)
232 {
233     struct device *dev;
234
235     dev = bus_find_device(bus, NULL, (void *)buf, driver_helper);
236     if (!dev)
237         return -ENODEV;
238     if (bus_rescan_devices_helper(dev, NULL) != 0)
239         return -EINVAL;
240     return count;
241 }

```

`store_drivers_probe` 这个函数一看就明白，其实就是调用了 `bus_find_device`，让这根总线上的设备都去找匹配自己的 `device_driver`。所以，每当我们写这个名为 `drivers_probe` 的文件，就是向内核发出命令，赶紧去把这个 `bus` 的设备和驱动遍历一遍，查看有没有能匹配的。

571 行到 578 行就是创建了一个名为 `drivers_autoprobe` 的文件，这个文件可读可写。文件的读操作对应于：

```

215 static ssize_t show_drivers_autoprobe(struct bus_type *bus, char *buf)
216 {
217     return sprintf(buf, "%d\n", bus->drivers_autoprobe);
218 }

```

就是返回我们刚才说的变量 `drivers_autoprobe`。文件的写操作对应于：

```

220 static ssize_t store_drivers_autoprobe(struct bus_type *bus,
221                                       const char *buf, size_t count)
222 {
223     if (buf[0] == '0')
224         bus->drivers_autoprobe = 0;
225     else
226         bus->drivers_autoprobe = 1;
227     return count;
228 }

```

把内容写到 `bus->drivers_autoprobe` 中。（只要写的内容不是 ‘0’，把 `bus->drivers_autoprobe`

置 1)。这样我们就可以在用户空间，通过读写 sysfs 中文件的方式，改变 drivers\_autoprobe 的值。从这点可以看出，sysfs 是用户与内核的接口，是用户与设备的接口，通过读写 sysfs 文件就能查看和改变内核的配置。

843 行，为 bus 的默认属性创建 sysfs 文件。好，bus\_register()搞定了。这个函数执行之后，会形成如图 2 所示的层次结构。

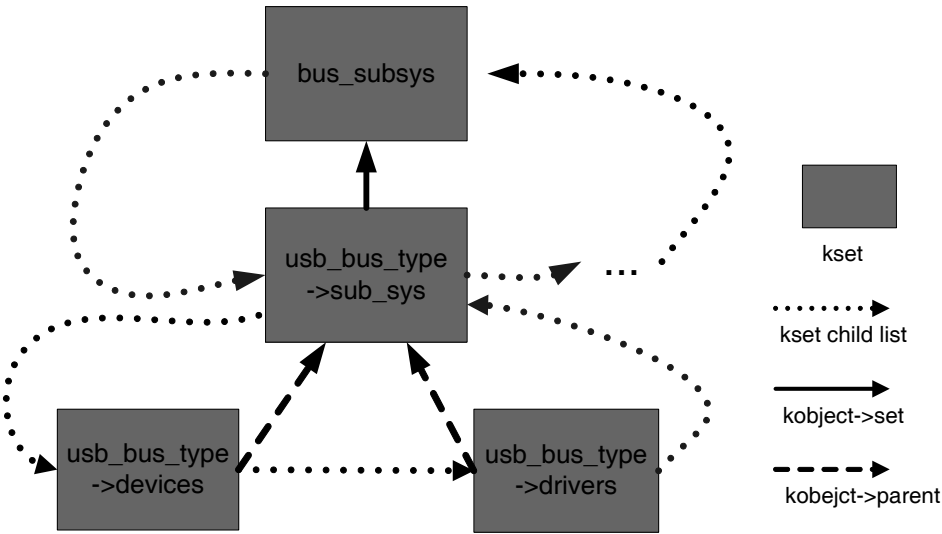


图 2 sysfs 层次结构

由于 bus\_subsys 对应于 /sysfs/bus/，故在 sysfs 文件系统中，会出现 /sysfs/bus/usb/ 这个目录。并且：

```
$ls /sys/bus/usb/  
devices/ drivers/ drivers_autoprobe drivers_probe
```

里面会有 devices 和 drivers 两个目录，和 drivers\_autoprobe，drivers\_probe 两个文件，与我们分析的一致。

2.4 示例二：usb storage 驱动

在 U 盘部分，分析了 U 盘的驱动。在此不妨回头查看 USB Storage 驱动是如何注册的。

```
static int __init usb_stor_init(void)  
{  
    int retval;  
    printk(KERN_INFO "Initializing USB Mass Storage driver...\n");  
  
    /* register the driver, return usb_register return code if error */  
    retval = usb_register(&usb_storage_driver);  
    if (retval == 0) {
```

```

        printk(KERN_INFO "USB Mass Storage support registered.\n");
        usb_usual_set_present(USB_US_TYPE_STOR);
    }
    return retval;
}

```

我们直接看 `usb_register(&usb_storage_driver)`, 这个 `usb_storage_driver` 是个 `usb_driver` 类型的变量, `usb_driver` 是专门针对 USB 设备所设置的变量的, 其中有个成员 `usb_storage_driver.drwrap.driver` 正是 `device_driver` 类型, 这个成员就是我们之前讨论的, 设备驱动层次结构的接口。接着往下看:

```

static inline int usb_register(struct usb_driver *driver)
{
    return usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME);
}

```

调用 `usb_register_driver()`, 定义如下:

```

734 int usb_register_driver(struct usb_driver *new_driver, struct module *owner,
735                        const char *mod_name)
736 {
737     int retval = 0;
738
739     if (usb_disabled())
740         return -ENODEV;
741
742     new_driver->drwrap.for_devices = 0;
743     new_driver->drwrap.driver.name = (char *) new_driver->name;
744     new_driver->drwrap.driver.bus = &usb_bus_type;
745     new_driver->drwrap.driver.probe = usb_probe_interface;
746     new_driver->drwrap.driver.remove = usb_unbind_interface;
747     new_driver->drwrap.driver.owner = owner;
748     new_driver->drwrap.driver.mod_name = mod_name;
749     spin_lock_init(&new_driver->dynids.lock);
750     INIT_LIST_HEAD(&new_driver->dynids.list);
751
752     retval = driver_register(&new_driver->drwrap.driver);
753
754     if (!retval) {
755         pr_info("%s: registered new interface driver %s\n",
756               usbcore_name, new_driver->name);
757         usbfs_update_special();
758         usb_create_newid_file(new_driver);
759     } else {
760         printk(KERN_ERR "%s: error %d registering interface "
761               "driver %s\n",
762               usbcore_name, retval, new_driver->name);
763     }
764
765     return retval;
766 }

```

742 行到 749 行, 老套路, 设置一些变量的值, 主要是设置两个函数 `usb_probe_interface`, `usb_unbind_interface`。这两个函数是在 `probe/remove` 设备时调用的, 我们待会就会看到。进入 752 行, `driver_register()`, 这就是向总线注册 USB Storage 驱动。我们其实可以大胆猜测, 所谓



注册驱动，把该驱动链入到 bus->drivers 中去。

```

153 int driver_register(struct device_driver * drv)
154 {
155     if ((drv->bus->probe && drv->probe) ||
156         (drv->bus->remove && drv->remove) ||
157         (drv->bus->shutdown && drv->shutdown)) {
158         printk(KERN_WARNING "Driver '%s' needs updating - please use
bus_type methods\n", drv->name);
159     }
160     klist_init(&drv->klist_devices, NULL, NULL);
161     return bus_add_driver(drv);
162 }

```

driver\_register 很简单，一顿 “check”，然后进入 bus\_add\_driver():

```

600 int bus_add_driver(struct device_driver *drv)
601 {
602     struct bus_type * bus = get_bus(drv->bus);
603     int error = 0;
604
605     if (!bus)
606         return -EINVAL;
607
608     pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
609     error = kobject_set_name(&drv->kobj, "%s", drv->name);
610     if (error)
611         goto out_put_bus;
612     drv->kobj.kset = &bus->drivers;
613     if ((error = kobject_register(&drv->kobj)))
614         goto out_put_bus;
615
616     if (drv->bus->drivers_autoprobe) {
617         error = driver_attach(drv);
618         if (error)
619             goto out_unregister;
620     }
621     klist_add_tail(&drv->knode_bus, &bus->klist_drivers);
622     module_add_driver(drv->owner, drv);
623
624     error = driver_add_attrs(bus, drv);
625     if (error) {
626         /* How the hell do we get out of this pickle? Give up */
627         printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n",
628                __FUNCTION__, drv->name);
629     }
630     error = add_bind_files(drv);
631     if (error) {
632         /* Ditto */
633         printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
634                __FUNCTION__, drv->name);
635     }
636
637     return error;
638 out_unregister:
639     kobject_unregister(&drv->kobj);
640 out_put_bus:
641     put_bus(bus);

```

```
642     return error;
643 }
```

612 行、613 行，把 `usb_storage_driver.drwrap.driver` 链入到 `kobject` 层次结构中去。我们得仔细地观察一下 `kobject_register()` 这个函数：

```
233 int kobject_register(struct kobject * kobj)
234 {
235     int error = -EINVAL;
236     if (kobj) {
237         kobject_init(kobj);
238         error = kobject_add(kobj);
239         if (!error)
240             kobject_uevent(kobj, KOBJ_ADD);
241     }
242     return error;
243 }
```

238 行，`kobject_add()` 以前见过，就是把这个 `kobj` 加入父 `kobj` 的链表中去。240 行，`kobject_uevent()`，这里引出了一个比较高级的机制。

回到 `bus_add_driver()`，616 行到 620 行，判断 USB 系统是否自动探测设备，如果不是，就什么也不干。否则，就调用 `driver_attach()` 去寻觅自己的另一半。`driver_attach()` 调用 trace 如下：

```
driver_attach()->bus_for_each_dev()->__driver_attach()
->driver_probe_device
```

通过 `bus->klist_devices` 链表遍历所有挂在 `bus` 上的设备，调用 `__driver_attach->driver_probe_device()` 去判定设备与驱动是否 `match`。

下面查看 `driver_probe_device()` 函数：

```
186 int driver_probe_device(struct device_driver * drv, struct device * dev)
187 {
188     int ret = 0;
189
190     if (!device_is_registered(dev))
191         return -ENODEV;
192     if (drv->bus->match && !drv->bus->match(dev, drv))
193         goto done;
194
195     pr_debug("%s: Matched Device %s with Driver %s\n",
196             drv->bus->name, dev->bus_id, drv->name);
197
198     ret = really_probe(dev, drv);
199
200 done:
201     return ret;
202 }
```

192 行，先判断该设备和驱动是否 `match`，此时调用的是 `usb_bus_type->usb_device_match()`，定义如下：

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)
```

```

541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545         /* interface drivers never match devices */
546         if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551
552     } else {
553         struct usb_interface *intf;
554         struct usb_driver *usb_drv;
555         const struct usb_device_id *id;
556
557         /* device drivers never match interfaces */
558         if (is_usb_device_driver(drv))
559             return 0;
560
561         intf = to_usb_interface(dev);
562         usb_drv = to_usb_driver(drv);
563
564         id = usb_match_id(intf, usb_drv->id_table);
565         if (id)
566             return 1;
567
568         id = usb_match_dynamic_id(intf, usb_drv);
569         if (id)
570             return 1;
571     }
572
573     return 0;
574 }

```

我们知道在 USB 系统里面，是一个接口对应一个 USB 设备驱动程序。这个函数的精髓在 564 行 `usb_match_id()`，实际上就是判断设备（接口）的基本信息（`idVendor`，`idProduct` 等），是否和驱动 USB Storage 的 `id_table`（征友条件）一致，如果匹配，直接返回。如果找不到，不着急，我们还有一次机会。568 行 `usb_match_dynamic_id()`，我们从该驱动的动态 `id_table` 中寻找，看能不能匹配得上。这样的话，驱动程序的 `id_table`（征友条件）就不是固定死板的了，而是可以动态增减的，增强了灵活性。那么如何动态更新驱动的动态 `id_table` 呢？不要着急，我们后面会讲到。没错，就是用 `sysfs` 文件系统作为借口。

好，看完了 `usb_device_match()`，再回到 `driver_probe_device`，如果没 `match`，直接返回。如果匹配上了，那么到 198 行，`really_probe()`。

```

100 static int really_probe(struct device *dev, struct device_driver *drv)
101 {
102     int ret = 0;
103
104     atomic_inc(&probe_count);
105     pr_debug("%s: Probing driver %s with device %s\n",
106             drv->bus->name, drv->name, dev->bus_id);
107     WARN_ON(!list_empty(&dev->devres_head));

```

```

108
109     dev->driver = drv;
110     if (driver_sysfs_add(dev)) {
111         printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
112                __FUNCTION__, dev->bus_id);
113         goto probe_failed;
114     }
115
116     if (dev->bus->probe) {
117         ret = dev->bus->probe(dev);
118         if (ret)
119             goto probe_failed;
120     } else if (drv->probe) {
121         ret = drv->probe(dev);
122         if (ret)
123             goto probe_failed;
124     }
125
126     driver_bound(dev);
127     ret = 1;
128     pr_debug("%s: Bound Device %s to Driver %s\n",
129             drv->bus->name, dev->bus_id, drv->name);
130     goto done;
131
132 probe_failed:
133     devres_release_all(dev);
134     driver_sysfs_remove(dev);
135     dev->driver = NULL;
136
137     if (ret != -ENODEV && ret != -ENXIO) {
138         /* driver matched but the probe failed */
139         printk(KERN_WARNING
140                "%s: probe of %s failed with error %d\n",
141                drv->name, dev->bus_id, ret);
142     }
143     /*
144      * Ignore errors returned by ->probe so that the next driver can try
145      * its luck.
146      */
147     ret = 0;
148 done:
149     atomic_dec(&probe_count);
150     wake_up(&probe_waitqueue);
151     return ret;
152 }

```

109 行和 110 行，由于已经找到了匹配的设备 and 驱动，那么把这二者关联起来。`driver_sysfs_add()`，互相在对方的 sysfs 目录下创建一个符号链接。

116 行到 126 行，调用 `probe` 函数，我们注意到 `usb_bus_type->probe` 是空的，所以实际 `probe` 的过程调用的是 `device_driver` 的 `probe` 函数，即在 `usb_register_driver` 函数 745 行设置的 `usb_probe_interface()` 函数，这个函数是一个 `wrapper`，它做了一堆 `check` 之后，调用 `usb_storage_driver->storage_probe()`。这时我们回到了 U 盘部分的“梦开始的地方”。

126 行，把设备链入驱动的设备列表 `klist_devices` 中。然后我们一路返回至 `bus_add_driver()`。

回到 `bus_add_driver()` 的 621 行，把驱动链入 `bus` 的驱动列表中。622 行的 `module_add_driver()`，把驱动在 `module` 的子系统 (`/sys/module/`) 中建立一个符号链接，同时在驱动下建立 `module` 的符号链接。624 行，把驱动的默认属性影射成 `sysfs` 的文件。625 行，`add_bind_files()`，创建 `bind`，`unbind` 两个文件。这两个文件也是 `sysfs` 暴露给用户的借口。我们来查看 `bind` 的定义：

```
static DRIVER_ATTR(bind, S_IWUSR, NULL, driver_bind);
```

`DRIVER_ATTR` 是一个宏，定义为：

```
#define DRIVER_ATTR(_name, _mode, _show, _store) \
struct driver_attribute driver_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

可以看出，`bind` 是只写文件，每次写这个文件意味着调用 `driver_bind()` 函数，`driver_bind` 定义在 `driver/base/bus.c` 中，作用是遍历 `bus` 上的所有设备，寻找自己的另一半。`unbind` 相反，调用 `driver_unbind` 把自己另一半给甩了，`bus_add_driver` 结束。

之后又回到 `usb_register_driver()`。757 行，`usbfs_update_special()` 更新 `usbfs` 的信息。`usbfs` 也是虚拟的文件系统，一般是挂载在 `/proc/bus/usb/`。758 行 `usb_create_newid_file`，这就是我们刚才讨论的驱动动态 `id_table`。Linux 把这个动态 `id` 表通过 `sysfs` 文件的形式暴露给用户。通过 `sysfs` 作为借口，这是一个很高明的设计。`usb_create_newid_file` 就是调用 `sysfs_create_file()` 创建一个文件，文件名是 `new_id`：

```
static DRIVER_ATTR(new_id, S_IWUSR, NULL, store_new_id);
```

文件是不可读的，只可写，写文件对应与 `usb_store_new_id`，就是把写的内容加入 `driver->dynid` 中。

好，现在我们注册了 `usb` 驱动 `usb_storage_driver`，`sysfs` 文件系统中，`/sys/bus/usb/driver` 下多了一个目录 `usb-storage`，并且：

```
$ ls /sys/bus/usb/drivers/usb-storage
bind new_id unbind module uevent
```

其中，`bind`，`new_id`，`unbind`，`uevent` 是文件，`module` 是链接。如果我们插了 U 盘的话，在 `usb-storage` 下还能看到这个设备。

---

## 3. sysfs 文件系统

现在，我们进入 `sysfs` 部分。`sysfs` 就是利用 `VFS` 的接口去读写 `kobject` 的层次结构，建立起来的文件系统。关于 `sysfs` 的内容就在 `fs/sysfs/` 下。`kobject` 的层次结构的更新与删除就是那些

XX\_register()们干的事情。

在 kobject\_add()里面，调用了 sysfs\_create\_dir()。让我们查看究竟是如何创建它的。

```

220 int sysfs_create_dir(struct kobject * kobj, struct dentry *shadow_parent)
221 {
222     struct dentry * dentry = NULL;
223     struct dentry * parent;
224     int error = 0;
225
226     BUG_ON(!kobj);
227
228     if (shadow_parent)
229         parent = shadow_parent;
230     else if (kobj->parent)
231         parent = kobj->parent->dentry;
232     else if (sysfs_mount && sysfs_mount->mnt_sb)
233         parent = sysfs_mount->mnt_sb->s_root;
234     else
235         return -EFAULT;
236
237     error = create_dir(kobj, parent, kobject_name(kobj), &dentry);
238     if (!error)
239         kobj->dentry = dentry;
240     return error;
241 }

```

当你看见这么些新东西，如 dentry 出现时，你一定感到很困惑。诚然，我一度为代码中突然出现的事物感到恐慌，人类对未知的恐惧是与生俱来的，面对死亡，面对怪力乱神，我们抱着一颗敬畏的心灵就可以了。而面对 Linux，我们始终坚信，未知肯定是可以被探索出来的。下面还是先介绍一下文件系统的基本知识。

### 3.1 文件系统

文件系统是个很模糊广泛的概念，“文件”狭义地说，是指磁盘文件，广义理解，可以有组织有次序地存储与任何介质（包括内存）的一组信息。Linux 把所有的资源都看成是文件，让用户通过一个统一的文件系统操作界面，也就是同一组系统调用，对属于不同文件系统的文件进行操作。这样，就可以对用户程序隐藏各种不同文件系统的实现细节，为用户程序提供了一个统一、抽象、虚拟的文件系统界面，这就是所谓“VFS（Virtual Filesystem Switch）”。这个抽象出来的接就是一组函数操作。

我们要实现一种文件系统就是要实现 VFS 所定义的一系列接口，file\_operations, dentry\_operations, inode\_operations 等，供上层调用。file\_operations 是对每个具体文件的读写操作，dentry\_operations, inode\_operations 则是对文件的属性，如改名字，建立或删除的操作。

```

struct file_operations {
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    ...

```

```
};

struct dentry_operations {
    ...
};

struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    ...
};
```

比如，我们写 C 程序，`open("hello.c", O_RDONLY)`，它通过系统调用的流程是这样的：

(1) `open()` -> /\*用户空间\*/ -> 系统调用->

(2) `sys_open()` -> `filp_open()` -> `dentry_open()` -> `file_operations.open()` /\*内核空间\*/

不同的文件系统，调用不同的 `file_operations.open()`，在 `sysfs` 下就是 `sysfs_open_file()`。

### 3.1.1 dentry & inode

我们在进程中要怎样去描述一个文件呢？我们用目录项（`dentry`）和索引节点（`inode`）。

```
struct dentry {
    struct inode *d_inode; /* Where the name belongs to - NULL is
    struct dentry *d_parent; /* parent directory */
    struct list_head d_child; /* child of parent list */
    struct dentry_operations *d_op;
    struct super_block *d_sb; /* The root of the dentry tree */
    void *d_fsdata; /* fs-specific data */
    unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /* small names */
    .....
};

struct inode {
    unsigned long i_ino;
    atomic_t i_count;
    umode_t i_mode;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    loff_t i_size;
    struct timespec i_atime;
    unsigned long i_blocks;
    unsigned short i_Bytes;
    unsigned char i_sock;
    struct inode_operations *i_op;
    struct file_operations *i_fop; /* former ->i_op->default_file_ops */
    struct super_block *i_sb;
    .....
};
```

所谓“文件”，就是按一定的形式存储在介质上的信息，所以一个文件其实包含了两方面的信息，一是存储的数据本身，二是有关该文件的组织和管理的信息。在内存中，每个文件都有一个 dentry（目录项）和 inode（索引节点）结构，dentry 记录着文件名、上级目录等信息，正是它形成了我们所看到的树状结构；而有关该文件的组织和管理的信息主要存放 inode 里面，它记录着文件在存储介质上的位置与分布。同时 dentry->d\_inode 指向相应的 inode 结构。dentry 与 inode 是多对一的关系，因为有可能一个文件有好几个文件名（硬链接, hard link）。

所有 dentry 用 d\_parent 和 d\_child 连接起来，就形成了我们熟悉的树状结构。

inode 代表的是物理意义上的文件，通过 inode 可以得到一个数组，这个数组记录了文件内容的位置，如该文件位于硬盘的第 3 块、第 8 块、第 10 块，那么这个数组的内容就是 3、8、10。其索引节点号 inode->i\_ino，在同一个文件系统中是唯一的，内核只要根据 i\_ino，就可以计算出它对应的 inode 在介质上的位置。就硬盘来说，根据 i\_ino 就可以计算出它对应的 inode 属于哪个块（block），从而找到相应的 inode 结构。但仅仅用 inode 还是无法描述出所有的文件系统，对于某一种特定的文件系统而言，比如 ext3，在内存中用 ext3\_inode\_info 描述。它是一个包含 inode 的“容器”。

```
struct ext3_inode_info {
    __le32 i_data[15];
    .....
    struct inode vfs_inode;
};
```

\_\_le32 i\_data[15]这个数组就是上一段中所提到的那个数组。

注意，在遥远的 2.4 内核，不同文件系统索引节点的内存映像(ext3\_inode\_info, reiserfs\_inode\_info, msdos\_inode\_info ...)都是用一个 union 内嵌在 inode 数据结构中的。但 inode 作为一种非常基本的数据结构而言，这样做太大了，不利于快速的分配和回收。但是后来发明了 container\_of(...)这种方法后，就把 union 移到了外部，我们可以用类似 container\_of(inode, struct ext3\_inode\_info, vfs\_inode)，从 inode 出发，得到其“容器”。

dentry 和 inode 终究都是在内存中的，它们的原始信息必须要有一个载体。否则断电之后岂不是完了？且听我慢慢道来。

文件可以分为磁盘文件、设备文件和特殊文件三种。设备文件暂且不介绍了。

### （1）磁盘文件

就磁盘文件而言，dentry 和 inode 的载体在存储介质（磁盘）上。对于像 ext3 这样的磁盘文件来说，存储介质中的目录项和索引节点载体如下：

```
struct ext3_inode {
    __le16 i_mode; /* File mode */
    __le16 i_uid; /* Low 16 bits of Owner Uid */
```



```
__le32 i_size; /* Size in Bytes */
__le32 i_atime; /* Access time */
__le32 i_ctime; /* Creation time */
__le32 i_mtime; /* Modification time */
__le32 i_dtime; /* Deletion Time */
__le16 i_gid; /* Low 16 bits of Group Id */
__le16 i_links_count; /* Links count */
.....
__le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
.....
}
struct ext3_dir_entry_2 {
    __u32 inode; /* Inode number */
    __u16 rec_len; /* Directory entry length */
    __u8 name_len; /* Name length */
    __u8 file_type;
    char name[EXT3_NAME_LEN]; /* File name */
};
__le32 i_block[EXT2 N BLOCKS]; /* Pointers to blocks */
```

i\_block 数组指示了文件的内容所存放的地点(在硬盘上的位置)。

ext3\_inode 是放在索引节点区，而 ext3\_dir\_entry 2 是以文件内容的形式存放在数据区的。我们只要知道了 ino，由于 ext3\_inode 大小已知，就可以计算出 ext3 inode 在索引节点区的位置 (ino \* sizeof(ext3\_inode))，而得到了 ext3\_inode，根据 i\_block 就可以知道这个文件的数据存放的地点。将磁盘上 ext3\_inode 的内容读入到 ext3\_inode\_info 中的函数是 ext3\_read\_inode()。以一个有 100 block 的硬盘为例，一个文件系统的组织布局大致如图 3 所示。位图区中的每一位表示每一个相应的对象有没有被使用。

(2) 特殊文件

特殊文件在内存中有 inode 和 dentry 数据结构，但是不一定在存储介质上有“索引节点”，它断电之后的确就完了，所以不需要什么载体。当从一个特殊文件读时，所读出的数据是由系统内部按一定的规则临时生成的，或从内存中收集、加工出来的。sysfs 里面就是典型的特殊文件。它存储的信息都是由系统动态地生成的，它动态包含了整个机器的硬件资源情况。从 sysfs 读写就相当于向 kobject 层次结构提取数据。

1	2	3	4-6	7-100
超级块	数据块位图	索引节点位图	索引节点区	数据区

图 3 文件系统的组织布局

还请注意，我们谈到目录项和索引节点时，有两种含义。一种是在存储介质（硬盘）中的 (如 ext3\_inode)，另一种是在内存中的，后者是根据在前者生成的。内存中的表示就是 dentry 和 inode，它是 VFS 中的一层，不管什么样的文件系统，最后在内存中描述它的都是 dentry 和

inode 结构。我们使用不同的文件系统，就是将它们各自的文件信息都抽象到 dentry 和 inode 中去。这样对于高层来说，我们就可以不关心底层的实现，我们使用的都是一系列标准的函数调用。这就是 VFS 的精髓，实际上就是面向对象。

我们在进程中打开一个文件 F，实际上就是要在内存中建立 F 的 dentry 和 inode 结构，并让它们与进程结构联系起来，把 VFS 中定义的接口给接起来。来看一看下面这个经典的图 4。

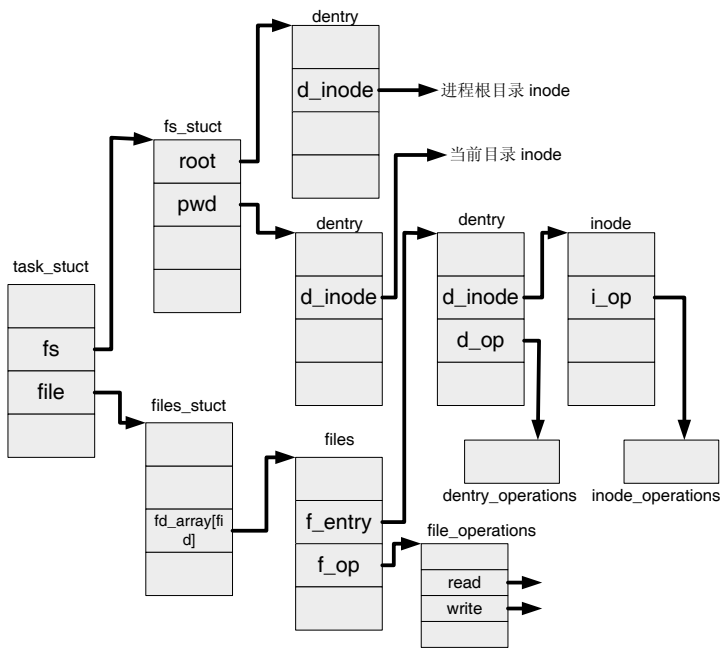


图 4 files\_struct、fs\_struct 与进程描述符的关系

3.1.2 一起散散步 path\_walk

前面说过，只要知道文件的索引节点号，就可以得到那个文件。但是在操作文件时，从没听说谁会拿着索引节点号来操作文件，只知道文件名而已。它们是如何“和谐”起来的呢？Linux 把目录也看成一种文件，里面记录着文件名与索引节点号的对应关系。比如在 EXT3 文件系统中，如果文件是一个目录，那么它的内容就是一系列 ext3\_dir\_entry\_2 的结构。

```
struct ext3_dir_entry_2 {
    __u32 inode; /* Inode number */
    __u16 rec_len; /* Directory entry length */
    __u8 name_len; /* Name length */
    __u8 file_type;
    char name[EXT3_NAME_LEN]; /* File name */
};
```

举一个例子，比如要打开/home/test/hello.c，首先，找到‘/’，读入其内容，找到名为“home”的文件的索引节点号，打开/home 这个“文件”，读入内容，找到名为“test”的文件的索引

节点号，同理，再打开文件“/home/test”，找到名为“hello.c”的文件的索引节点号，最后就得到/home/test/hello.c了。这就是 path\_walk()函数的原理。

其中，根据一个文件夹的 inode，和一个文件名来获取该文件的 inode 结构的函数，就叫 lookup，它是 inode operations 里面的函数。lookup，顾名思义，就是查找，比如在 test 这个文件夹下，查找有没有叫 hello.c 的文件，如果有，就从存储介质中读取其 inode 结构，并用 dentry->d\_inode 指向它。所以，只要知道了文件的路径和名字，总可以从根目录开始，一层一层地往下走，定位到某一个文件。

### 3.1.3 super\_block 与 vfsmount

接下来还要介绍两个数据结构，super\_block 和 vfsmount。super\_block 结构是从所有具体的文件系统所抽象出来的一个结构，每一个文件系统实例都会有一对应的 super\_block 结构。比如每一个 ext2 的分区就有一个 super\_block 结构，它记录了该文件系统实例（分区）的某些描述性的信息，比如该文件系统实例的文件系统类型，有多大，磁盘上每一块的大小，还有就是 super\_operations。它与 inode，dentry 一样，只是某些内容在内存中的映像。就 ext2 文件系统而言，设备上的超级块为 ext2\_super\_block。由于 sysfs 是虚拟的文件系统，独一无二，并且只能被“mount”一次，sysfs 的 super\_block 结构是 sysfs\_sb，sysfs\_sb 也是动态地从内存中生成的。

还有要提一下 super\_operations，它也算是 VFS 的一个接口。实现一个文件系统时，file\_operations，dentry\_operations，inode\_operations 和 super\_operations 这四个结构都要实现。

把一个设备安装到一个目录节点时要用一个 vfsmount 作为连接件。

```
struct vfsmount {
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent; /* fs we are mounted on */
    struct dentry *mnt_mountpoint; /* dentry of mountpoint */
    struct dentry *mnt_root; /* root of the mounted tree */
    struct super_block *mnt_sb; /* pointer to superblock */
    .....
};
```

对于某个文件系统实例，内存中 super\_block 和 vfsmount 都是唯一的。比如，我们将挂载某个硬盘分区：

```
mount -t vfat /dev/hda2 /mnt/d
```

实际上就是新建一个 vfsmount 结构作为连接件，vfsmount->mnt\_sb = /dev/hda2 的超级块结构；vfsmount->mntroot = /dev/hda2 的“根”目录的 dentry；vfsmount->mnt\_mountpoint = /mnt/d 的 dentry；vfsmount->mnt\_parent = /mnt/d 所属的文件系统的 vfsmount，并且把这个新建的 vfsmount 连入一个全局的 hash 表 mount hashtable 中。

从而我们就可以从总根‘/’开始，沿着“dentry”往下找。假如碰到某个目录的 dentry 是被“mount”了的，那么就从 mount\_hashtable 表中去寻找相应的 vfsmount 结构（函数是

lookup\_mnt())。然后我们得到 `vfsmount->mnt_root`，就可以找到 `mount` 在该目录的文件系统的“根”`dentry` 结构。然后又继续往下走，就可以畅通无阻了。

关于 `path walk()` 的代码我就不“贴”了，太长了。其实懂了原理后再去看，很简单。我当年就是看完这个函数后，信心倍增。`path_walk`，不管前面是高速公路，或是泥泞的乡间小路，我们都要走到底。

## 3.2 sysfs

Linus 曾经炮轰 C++，“C++ 是一种糟糕的 (horrible) 语言。而且因为有大量不够标准的程序员在使用使许多真正懂得底层问题，而不会折腾那些白痴‘对象模型’”。牛人就是牛气冲天。

在 `fs/sysfs/` 下面，除去 `Makefile`，还有 8 个文件。其中，`bin.c`、`file.c`、`dir.c`、`symlink.c` 分别代表了在 `sysfs` 文件系统中当文件类型为二进制文件、普通文件、目录、符号连接时的各自 `file_operations` 结构体的实现。`inode.c` 则是 `inode_operations` 的实现，还有创建和删除 `inode`。`mount.c` 包括了 `sysfs` 的初始化函数。`sysfs.h` 就是头文件，里面有函数的原形，并将其“extern”出去。

`sysfs` 的文件系统的所读写的信息是存放在 `kobject` 当中，那么 `dentry` 是如何与 `kobject` 联系起来的呢？是通过 `sysfs_dirent`。

### 3.2.1 sysfs\_dirent

`sysfs` 文件系统有自己的 `dirent` 结构，`dirent = directory_entry`（目录实体）。每一个 `dentry` 对应了一个 `dirent` 结构，`dentry->d_fsdata` 是一个 `void` 的指针，由它指向不同的文件系统所特有 `dirent` 结构。在 `sysfs` 中，则指向 `sysfs_dirent`，定义为：

```
struct sysfs_dirent {
    atomic_t s_count;
    struct list_head s_sibling;
    struct list_head s_children;
    void * s_element;
    int s_type;
    umode_t s_mode;
    struct dentry * s_dentry;
    struct iattr * s_iattr;
    atomic_t s_event;
};
```

`s_count` 是引用计数，`s_sibling`、`s_children` 指针使这些 `sysfs_dirent` 连成一个树状结构。`s_type` 则说明了这个 `dirent` 具体的类型：

```
#define SYSFS_ROOT 0x0001
#define SYSFS_DIR 0x0002
#define SYSFS_KOBJ_ATTR 0x0004
#define SYSFS_KOBJ_BIN_ATTR 0x0008
#define SYSFS_KOBJ_LINK 0x0020
```

`s_element` 就是指向相应与 `s_type` 类型相对应的数据结构, 如 `DIR` (就是 `kobject`, 一个 `kobject` 对应一个 `DIR`), `KOBJ_ATTR` (attribute 属性, 代表一个文件)。

`sysfs_dirent` 是 `kobject` 和 `sysfs` 联系的一个中间连接结构。它通过 `s_sibling`, `s_children` 连接成一个层次结构。而且它的层次结构与 `sysfs` 完全一致的, 它就是一个连接 `kobject` 和 `dentry` 结构的连接件。

对于 `sysfs` 下的文件夹而言, `dentry`, `dirent`, `kobject` 之间通过指针相互联系起来。

```
dentry->d_fsdata = &dirent;
dirent->element = &kobject;
kobject->dentry = &dentry;
```

### 3.2.2 sysfs\_create\_dir()

每当我们新增一个 `kobject` 结构时, 同时会在 `/sys` 下创建一个目录。

```
kobject_add() -> create_dir() -> sysfs_create_dir()
```

此时, 我还想重申, 内核代码的更新换代很快, 我们的目的是懂得代码背后的原理、知识, 或曰哲学。我不想讲得太细, 因为关于 `sysfs` 的部分从 2.6.22 到现在已经改了很多了。但其总体架构没变。写此文的目的是让您跟着我的思路走一遍, 对 `sysfs` 有了一个总体上的认识。然后自己就可以去看最新的代码了。最新的代码肯定是效率更高、条理逻辑更清晰。

`sysfs_create_dir()`流程图如下:

```
sysfs_create_dir()
-> create_dir()
    -> *d = sysfs_get_dentry()
        -> lookup_hash()
            -> __lookup_hash()
                -> cached_lookup()
                    -> new = d_alloc(base, name);
                        -> inode->i_op->lookup(inode, new, nd)
        -> sysfs_create(*d, mode, init_dir)
            -> sysfs_new_inode(mode)
                -> init_dir(inode); \\Call back function
        -> sysfs_make_dirent()
            -> sysfs_new_dirent()
                -> dentry->d_fsdata = sysfs_get(sd);
                -> dentry->d_op = &sysfs_dentry_ops;
        -> (*d)->d_op = &sysfs_dentry_ops;
```

#### (1) sysfs\_get\_dentry()分析

`sysfs_get_dentry()`作用是根据父辈 `dentry` 和文件名得到 `dentry` 结构。首先在缓存中找, 如果找到就返回, 找不到就用 `d_alloc()`新建一个 `dentry` 结构。接着调用 `lookup` 函数, 它定义如下:

```
struct inode_operations sysfs_dir_inode_operations = {
    .lookup = sysfs_lookup,
};
```

```
static struct dentry * sysfs_lookup(struct inode *dir, struct dentry *dentry,
                                   struct nameidata *nd)
{
    struct sysfs_dirent * parent_sd = dentry->d_parent->d_fsdata;
    struct sysfs_dirent * sd;
    int err = 0;

    list_for_each_entry(sd, &parent_sd->s_children, s_sibling) {
        if (sd->s_type & SYSFS_NOT_PINNED) {
            const unsigned char * name = sysfs_get_name(sd);

            if (strcmp(name, dentry->d_name.name))
                continue;

            if (sd->s_type & SYSFS_KOBJ_LINK)
                err = sysfs_attach_link(sd, dentry);
            else
                err = sysfs_attach_attr(sd, dentry);
            break;
        }
    }

    return ERR_PTR(err);
}
```

前面讲过 `lookup` 函数的作用。它在 `inode` 代表的文件夹下查找有没有名为 `dentry.dname.name` 的文件。如果有，就将其对应的 `inode` 结构从信息的载体中读出来。由于是新建的文件夹，所以 `lookup` 函数在我们这个故事里根本没做事。但是还是忍不住想分析一下 `lookup` 函数。

`sysfs` 文件系统中，目录的 `inode` 和 `dentry` 结构一直都是存在于内存中的，所以不用再进行读取了。而文件，链接的 `inode` 事先是没有的，需要从载体中读出。

但是 `sysfs` 的 `lookup` 还有它不同之处。其他文件系统像 `ext3`，普通文件的 `inode` 在文件创建之时，就已经创建了。但是 `sysfs` 不一样，它在创建普通文件时，只是先创建一个 `sysfs_dirent` 结构。创建 `inode` 的工作是推迟到 `lookup` 函数来完成的。在下一节 `sysfs_create_file()` 会看到这一点。

`sysfs_attach_attr()` 和 `sysfs_attach_link()` 的作用就是根据 `dentry` 和 `sysfs_dirent` 新建一个 `inode`。

总之，我们通过 `sysfs_get_dentry()` 得到了一个新建的 `dentry` 结构。

## (2) `sysfs_create()` 分析

```
sysfs_create()->sysfs_new_inode(mode)->new_inode(sysfs_sb)
```

创建一个新的索引节点 `inode`，`sysfs_sb` 是 `sysfs` 的超级块 (`super_block`) 结构。`mode` 则是 `inode` 的属性，它记录了如下信息，比如，文件类型（是文件夹、链接还是普通文件），`inode` 的所有者，创建时间等。

(3) sysfs\_make\_dirent()分析

至此，我们得到了一个 dirent 结构，先初始化，再把它连接到上层目录的 sysfs\_dirent 的 s\_children 链表里去。sysfs\_make\_dirent()为刚刚新建出来的 dentry 建立一个 dirent 结构。并将 dentry 和 dirent 联系起来。

(4) 总结

在 sysfs 下创建一个目录，提供的函数是 sysfs\_create\_dir()。创建了 dentry，dirent，inode 结构，它们之间的关系见图 5。

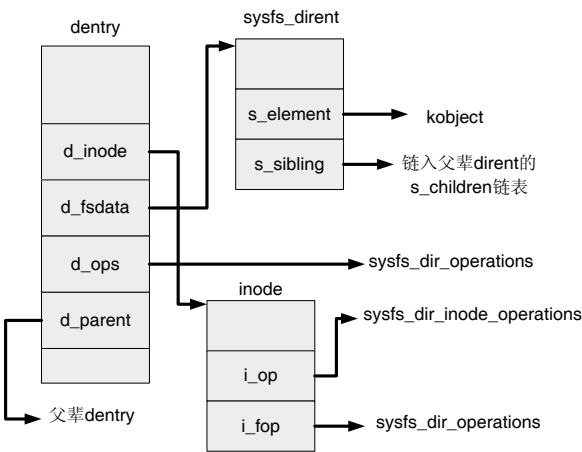


图 5 dentry, dirent, inode 关系

3.2.3 sysfs\_create\_file()

```
sysfs_create_dir()
-> sysfs_add_file()
    -> sysfs_make_dirent()
        -> sysfs_new_dirent()
```

sysfs 文件系统中，普通文件对应于 kobject 中的属性。用 sysfs\_create\_file(), 参数如下：

```
sysfs_create_file (struct kobject * kobj, const struct attribute * attr)
```

传给它的参数是 kobj 和 attr，其中，kobject 对应的是文件夹，attribute 对应的是该文件夹下的文件。

```
int sysfs_create_file(struct kobject * kobj, const struct attribute * attr)
{
    BUG_ON(!kobj || !kobj->dentry || !attr);
    return sysfs_add_file(kobj->dentry, attr, SYSFS_KOBJ_ATTR);
}
```

它直接调用 sysfs\_add\_file():

```

int sysfs_add_file(struct dentry * dir, const struct attribute * attr, int type)
{
    struct sysfs_dirent * parent_sd = dir->d_fsdata;
    umode_t mode = (attr->mode & S_IALLUGO) | S_IFREG;
    int error = 0;
    down(&dir->d_inode->i_sem);
    error = sysfs_make_dirent(parent_sd, NULL, (void *) attr, mode, type);
    up(&dir->d_inode->i_sem);
    return error;
}

int sysfs_make_dirent(struct sysfs_dirent * parent_sd, struct dentry * dentry,
void * element, umode_t mode, int type)
{
    struct sysfs_dirent * sd;
    sd = sysfs_new_dirent(parent_sd, element);
    if (!sd)
        sd->s_mode = mode;
        sd->s_type = type;
        sd->s_dentry = dentry;
        if (dentry) {
            dentry->d_fsdata = sysfs_get(sd);
            dentry->d_op = &sysfs_dentry_ops;
        }
    return 0;
}

```

sysfs\_create\_file() 仅仅是调用了 sysfs\_make\_dirent() 创建 sysfs\_dirent 结构。与 sysfs\_create\_file ()不同，它甚至没有在 sysfs 文件系统下创建 inode 结构。这项工作被滞后了，在 sysfs\_lookup()->sysfs\_attach\_attr()里面完成。

### 3.3 file\_operations

前面说到，如何创建文件夹和文件。我们发现，在 sysfs 中，inode 并不那么重要。这是因为我们所要读写的信息已经就在内存中，并且已经形成了层次结构。我们只需有 dentry，就可以 dentry->fsdata，就能找到我们读些信息的来源——sysfs\_dirent 结构。这也是我觉得有必要研究 sysfs 的原因之一，因为它简单，而且不涉及具体的硬件驱动，但是从这个过程中，我们可以把文件系统中的一些基本数据结构搞清楚。接下来，以读取 sysfs 文件和文件夹的内容为例子，讲一讲文件读的流程。那么关于写，还有关于 symlink 完全可以依此类推了。

文件类型分别为文件夹和普通文件时的 file\_operations 如下。

```

struct file_operations sysfs_dir_operations = {
    .open = sysfs_dir_open,
    .release = sysfs_dir_close,
    .llseek = sysfs_dir_llseek,
    .read = generic_read_dir,
    .readdir = sysfs_readdir,
};

struct file_operations sysfs_file_operations = {
    .read = sysfs_read_file,
    .write = sysfs_write_file,
    .llseek = generic_file_llseek,
    .open = sysfs_open_file,
}

```



```
.release = sysfs_release,
};
```

### 3.3.1 示例一：读入 sysfs 目录的内容

新建文件夹时，设置了：

```
inode->i_op = &sysfs_dir_inode_operations;
inode->i_fop = &sysfs_dir_operations;
```

用一个非常简短的程序来测试：

```
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
int main(){
    DIR * dir;
    struct dirent *ptr;
    dir = opendir("/sys/bus/");
    while((ptr = readdir(dir))!=NULL){
        printf("d_name :%s\n",ptr->d_name);
    }
    closedir(dir);
    return -1;
}
```

在用户空间，用 gcc 编译执行即可。我们来查看它究竟做了什么。

#### (1) sysfs\_dir\_open()。

这是个用户空间的程序。opendir()是 glibc 的函数，glibc 也就是著名的标准 C 库。至于 opendir()是如何与 sysfs\_dir\_open()接上头的，那还得去看 glibc 的代码。这里就不分析了。glibc 可以从 gnu 的网站上自己下载源代码，编译。再用 gdb 调试，就可以看得很清楚。

opendir()的函数流程如下：

```
opendir("/sys/bus/") -> 系统调用 -> sys_open() -> filp_open() -> dentry_open() ->
sysfs_dir_open()
```

sysfs\_dir\_open()定义为：

```
static int sysfs_dir_open(struct inode *inode, struct file *file)
{
    struct dentry * dentry = file->f_dentry;
    struct sysfs_dirent * parent_sd = dentry->d_fsdata;
    down(&dentry->d_inode->i_sem);
    file->private_data = sysfs_new_dirent(parent_sd, NULL);
    up(&dentry->d_inode->i_sem);
    return file->private_data ? 0 : -ENOMEM;
}
```

- 内核空间：新建一个 dirent 结构，将它地址保存在 file->private\_data 中。
- 用户空间：新建了一个 DIR 结构，DIR 结构如下。

```
#define __dirstream DIR
struct __dirstream
{
    int fd; /* File descriptor. */
    char *data; /* Directory block. */
    size_t allocation; /* Space allocated for the block. */
    size_t size; /* Total valid data in the block. */
    size_t offset; /* Current offset into the block. */
    off_t filepos; /* Position of next entry to read. */
    __libc_lock_define (, lock) /* Mutex lock for this structure. */
};
```

## (2) sysfs\_readdir()。

readdir 函数调用流程如下：

```
readdir(dir) -> getdents() -> 系统调用 -> sys32_readdir() -> vfs_readdir() ->
sysfs_readdir()
```

readdir 函数有点儿复杂，虽然在 main 函数中的 while 循环中，readdir 被执行了多次，查看 glibc 里面的代码：

```
readdir(dir){
    .....
    if (dirp->offset >= dirp->size){
        .....
        getdents()
        .....
    }
    .....
}
```

实际上，getdents() -> ... -> sysfs\_readdir() 只被调用了两次，getdents() 一次就把所有的内容都读完，存在 DIR 结构当中，readdir() 只是从 DIR 结构当中每次取出一个。DIR(dirstream) 结构就是一个流。而回调函数 filldir 的作用就是往这个流中填充数据。第二次调用 getdents() 是用户把 DIR 里面的内容读完了，所以它又调用 getdents()，但是这次 getdents() 会返回 NULL。

```
static int sysfs_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
    struct dentry *dentry = filp->f_dentry;
    struct sysfs_dirent * parent_sd = dentry->d_fsdata;
    struct sysfs_dirent *cursor = filp->private_data;
    struct list_head *p, *q = &cursor->s_sibling;
    ino_t ino;
    int i = filp->f_pos;
    switch (i) {
    case 0:
        ino = dentry->d_inode->i_ino;
        if (filldir(dirent, ".", 1, i, ino, DT_DIR) < 0)
            break;
        filp->f_pos++;
        i++;
        /* fallthrough */
    case 1:
        ino = parent_ino(dentry);
```

```

        if (filldir(dirent, "..", 2, i, ino, DT_DIR) < 0)
            break;
        filp->f_pos++;
        i++;
        /* fallthrough */
default:
    if (filp->f_pos == 2) {
        list_del(q);
        list_add(q, &parent_sd->s_children);
    }
    for (p=q->next; p!= &parent_sd->s_children; p=p->next) {
        struct sysfs_dirent *next;
        const char * name;
        int len;
        next = list_entry(p, struct sysfs_dirent,
                          s_sibling);
        if (!next->s_element)
            continue;
        name = sysfs_get_name(next);
        len = strlen(name);
        if (next->s_dentry)
            ino = next->s_dentry->d_inode->i_ino;
        else
            ino = iunique(sysfs_sb, 2);
        if (filldir(dirent, name, len, filp->f_pos, ino,
                    dt_type(next)) < 0)
            return 0;
        list_del(q);
        list_add(q, p);
        p = q;
        filp->f_pos++;
    }
    return 0;
}

```

看 `sysfs_readdir()` 其实很简单, 它就是从我们调用 `sysfs_dir_open()` 时新建的一个 `sysfs_dirent` 结构开始, 遍历当前 `dentry->dirent` 下的所有子 `sysfs_dirent` 结构。读出名字, 再回调函数 `filldir()` 将文件名、文件类型等信息, 按照一定的格式写入某个缓冲区。一个典型的 `filldir()` 就是 `filldir64()`, 它的作用是按一定格式向缓冲区写数据, 再把数据复制到用户空间去。

### 3.3.2 示例二：读入 sysfs 普通文件的内容

同样, 看下面这个小程序:

```

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(){
    char *name = "/sys/bus/usb/drivers_autoprobe ";
    char buf[1000];
    int fd;
    int size;
    fd = open(name, O_RDONLY);
    printf("fd:%d\n", fd);
    size = read(fd, buf, sizeof(buf));
}

```

```

    printf("size:%d\n",size);
    printf("%s",buf);
    close(fd);
    return -1;
}

```

(1) sysfs\_open\_file()。

open()的调用流程如下:

```

open() -> 系统调用 -> sys_open() -> filp_open()-> dentry_open() ->
sysfs_open_file()

```

sysfs\_open\_file()定义为:

```

static int sysfs_open_file(struct inode * inode, struct file * filp)
{
    return check_perm(inode, filp);
}

```

check\_perm ()检查一下权限, 创建一个 sysfs 的缓冲区 sysfs\_buffer buffer, 并设置其 sysfs\_ops\_sysfs\_buffer->ops。在我们这个故事里, sysfs\_buffer->ops 被设置成 bus\_sysfs\_ops。最后让 file->private\_data = buffer。

```

struct sysfs_buffer {
    size_t count;
    loff_t pos;
    char * page;
    struct sysfs_ops * ops;
    struct semaphore sem;
    int needs_read_fill;
};

```

(2) sysfs\_read\_file()。

read()的调用流程如下:

```

read() -> 系统调用 -> sys_read() -> vfs_read() -> sysfs_read_file()

```

sysfs\_read\_file()定义为:

```

sysfs_read_file(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    struct sysfs_buffer * buffer = file->private_data;
    ssize_t retval = 0;
    down(&buffer->sem);
    if (buffer->needs_read_fill) {
        if ((retval = fill_read_buffer(file->f_dentry,buffer)))
            goto out;
    }
    pr_debug("%s: count = %d, ppos = %lld, buf = %s\n",
        __FUNCTION__, count, *ppos, buffer->page);
    retval = flush_read_buffer(buffer, buf, count, ppos);
out:
    up(&buffer->sem);
}

```

```

        return retval;
    }

static int fill_read_buffer(struct dentry * dentry, struct sysfs_buffer * buffer)
{
    struct attribute * attr = to_attr(dentry);
    struct kobject * kobj = to_kobj(dentry->d_parent);
    struct sysfs_ops * ops = buffer->ops;
    int ret = 0;
    ssize_t count;
    if (!buffer->page)
        buffer->page = (char *) get_zeroed_page(GFP_KERNEL);
    if (!buffer->page)
        return -ENOMEM;
    count = ops->show(kobj, attr, buffer->page);
    buffer->needs_read_fill = 0;
    BUG_ON(count > (ssize_t)PAGE_SIZE);
    if (count >= 0)
        buffer->count = count;
    else
        ret = count;
    return ret;
}

static int flush_read_buffer(struct sysfs_buffer * buffer, char __user * buf,
size_t count, loff_t * ppos)
{
    int error;
    if (*ppos > buffer->count)
        return 0;
    if (count > (buffer->count - *ppos))
        count = buffer->count - *ppos;
    error = copy_to_user(buf, buffer->page + *ppos, count);
    if (!error)
        *ppos += count;
    return error ? -EFAULT : count;
}

```

顺着 sysfs\_read\_file () 往下走，函数调用流程为：

```

sysfs_read_file()
-> fill_read_buffer()
    -> sysfs_buffer->bus_sysfs_ops->bus_attr_show()
        -> bus_attribute->show_bus_version()
    -> flush_read_buffer()

```

fill\_read\_buffer() 是真正的“读”，它把内容读到缓冲区 sysfs\_buffer，flush\_read\_buffer() 把缓冲区复制到用户空间。

## 反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036